Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors

by

Gerhard Schmidt

http://www.avr-asm-tutorial.net

May 2016

History:

Corrected a calculation error in May 2016

Added page on assembler concept in February 2011

Added chapter on code structures in April 2009

Additional corrections and updates as of January 2008

Corrected version as of July 2006

Original version as of December 2003

Content

1 Why learning Assembler?

Assembler or other languages, that is the question. Why should I learn another language, if I already learned other programming languages? The best argument: while you live in France you are able to get through by speaking English, but you will never feel at home then, and life remains complicated. You can get through with this, but it is rather inappropriate. If things need a hurry, you should use the country's language.

Many people that are deeper into programming AVRs and use higher-level languages in their daily work recommend that beginners start with learning assembly language. The reason is that sometimes, namely in the following cases:

- if bugs have to be analyzed,
- if the program executes different than designed and expected,
- if the higher-level language doesn't support the use of certain hardware features,
- if time-critical in line routines require assembly language portions,

it is necessary to understand assembly language, e. g. to understand what the higher-level language compiler produced. Without understanding assembly language you do not have a chance to proceed further in these cases.

Short and easy

Assembler instructions translate one by one to executed machine instructions. The processor needs only to execute what you want it to do and what is necessary to perform the task. No extra loops and unnecessary features blow up the generated code. If your program storage is short and limited and you have to optimize your program to fit into memory, assembler is choice 1. Shorter programs are easier to debug, every step makes sense.

Fast and quick

Because only necessary code steps are executed, assembly programs are as fast as possible. The duration of every step is known. Time critical applications, like time measurements without a hardware timer, that should perform excellent, must be written in assembler. If you have more time and don't mind if your chip remains 99% in a wait state type of operation, you can choose any language you want.

Assembler is easy to learn

It is not true that assembly language is more complicated or not as easy to understand than other languages. Learning assembly language for whatever hardware type brings you to understand the basic concepts of any other assembly language dialects. Adding other dialects later is easy. As some features are hardware-dependent optimal code requires some familiarity with the hardware concept and the dialect. What makes assembler sometimes look complicated is that it requires an understanding of the controller's hardware functions. Consider this an advantage: by learning assembly language you simultaneously learn more about the hardware. Higher level languages often do not allow you to use special hardware features and so hide these functions.

The first assembly code does not look very attractive, with every 100 additional lines programmed it looks better. Perfect programs require some thousand lines of code of exercise, and optimization requires lots of work. The first steps are hard in any language. After some weeks of programming you will laugh if you go through your first code. Some assembler instructions need some months of experience.

AVRs are ideal for learning assembler

Assembler programs are a little bit silly: the chip executes anything you tell it to do, and does not ask you if you are sure overwriting this and that. All protection features must be programmed by you, the chip does exactly anything like it is told, even if it doesn't make any sense. No window warns you, unless you programmed it before.

To correct typing errors is as easy or complicated as in any other language. Basic design errors, the more tricky type of errors, are also as complicated to debug like in any other computer language. But: testing programs on ATMEL chips is very easy. If it does not do what you expect it to do, you can easily add some diagnostic lines to the code, reprogram the chip and test it. Bye, bye to you EPROM programmers, to the UV lamps used to erase your test program, to you pins that don't fit into the socket after having them removed some dozen times.

Changes are now programmed fast, compiled in no time, and either simulated in the studio or checked in-circuit. No pin is removed, and no UV lamp gives up just in the moment when you had your excellent idea about that bug.

Test it!

Be patient doing your first steps! If you are familiar with another (high-level) language: forget it for the first time. Behind every assembler language there is a certain hardware concept. Most of the special features of other computer languages don't make any sense in assembler.

The first five instructions are not easy to learn, after that your learning speed rises fast. After you had your first lines: grab the instruction set list and lay back in the bathtub, wondering what all the other instructions are like.

Serious warning: Don't try to program a mega-machine to start with. This does not make sense in any computer language, and just produces frustration. Start with the small „Hello world“-like examples, e. g. turning some LEDs on and off for a certain time, then explore the hardware features a bit deeper.

Recommendation: Comment your subroutines and store them in a special directory, if debugged: you will need them again in a short time.

Have success!

2 The concept behind the language assembler in micro-controllers

Attention! These pages are on programming micro-controllers, not on PCs with Linux- or Windows operating systems and similar elephants, but on a small mice. It is not on programming Ethernet mega-machines, but on the question why a beginner should start with assembler and not with a complex high-level language.

This page shows the concept behind assembler, what those familiar with high-level languages have to give up to learn assembler and why assembler is not machine language.

2.1 The hardware of micro-controllers

What has the hardware to do with assembler? Much, as can be seen from the following.

The concept behind assembler is to make the hardware resources of the processor accessible. Resources means all hardware components, like

* the central processing unit (CPU) and its math servant, the arithmetic and logic unit (ALU),
* the diverse storage units (internal and external RAM, EEPROM storage),
* the ports that control characteristics of port-bits, timers, AD converters, and other devices.

Accessible means directly accessible and not via drivers or other interfaces, that an operating system provides. That means, you control the serial interface or the AD converter, not some other layer between you and the hardware. As award for your efforts, the complete hardware is at your command, not only the part that the compiler designer and the operating system programmer provides for you.

2.2 How the CPU works

Most important for understanding assembler is to understand how the CPU works. The CPU reads instructions (instruction fetch) from the program storage (the flash), translates those into executable steps and executes those. In AVRs, those instructions are written as 16 bit numbers to the flash storage, and are read from there (first step). The number read then translates (second step) e. g. to transporting the content of the two registers R0 and R1 to the ALU (third step), to add those (fourth step) and to write the result into the register R0 (fifth step). Registers are simple 8 bit wide storages that can directly be tied to the ALU to be read from and to be written to.

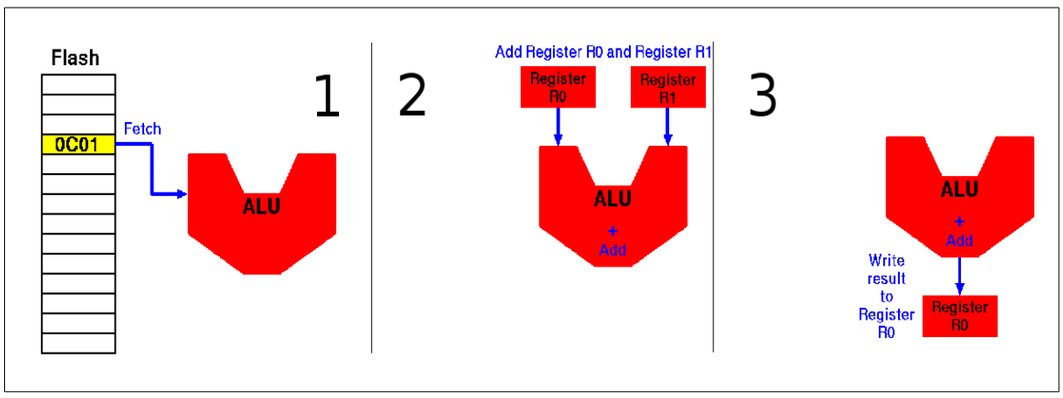
The coding of instructions is demonstrated by some examples.

CPU operation	Code (binary)	Code (hex)
Send CPU to sleep	1001.0101.1000.1000	9588
Add register R1 to register R0	0000.1100.0000.0001	0C01
Subtract register R1 from register R0	0001.1000.0000.0001	1801
Write constant 170 to register R16	1110.1010.0000.1010	EA0A
Multiply register R3 with register R2 and write the result to registers R1 (MSB) and R0 (LSB)	1001.1100.0011.0010	9C32

So, if the CPU reads hex 9588 from the flash storage, it stops its operation and does not fetch instructions any more. Don't be afraid, there is another mechanism necessary before the CPU executes this. And you can wake up the CPU from that.

Executing instructions

If the CPU reads hex 0C01, R0 and R1 is added and the result is written to register R0. This is executed like demonstrated in the picture.



First the instruction word (16 bit) is read from the flash and translated to executable steps (1).

The next step connects the registers to the ALU inputs, and adds their content (2).

Next, the result is written to the register (3).

If the CPU reads hex 9C23 from the flash, the registers R3 and R2 are muliplied and the result is written to R1 (upper 8 bits) and R0 (lower 8 bits). If the ALU is not equipped with hardware for multiplication (e. g. in an ATtiny13), the 9C23 does nothing at all. It doesn't even open an error window (the tiny13 doesn't have that hardware)!

In principle the CPU can execute 65,536 (16-bit) different instructions. But because not only 170 should be written to a specific register, but values between 0 and 255 to any register between R16 and R31, this load instruction requires 256*16 = 4,096 of the 65,536 theoretically possible instructions. The direct load instruction for the constant c (c7..c0) and registers r (r3..r0, r4 is always 1 and not encoded) is coded like this:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDI R,C	1	1	1	0	c7	c6	c5	c4	r3	r2	r1	r0	c3	c2	c1	c0

Why those bits are placed like this in the instruction word remains ATMEL's secret.

Addition and subtraction require 32*32 = 1,024 combinations and the target registers R0..R31 (t4..t0) and source registers R0..R31 (s4..s0) are coded like this:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD Rt,Rs	0	0	0	0	1	1	s4	t4	t3	t2	t1	t0	s3	s2	s1	S0
SUB Rt,Rs	0	0	0	1	1	0	s4	t4	t3	t2	t1	t0	s3	s2	s1	s0

Please, do not learn these bit placements, you will not need them later. Just understand how an instruction word is coded and executed.

2.3 Instructions in assembler

There is no need to learn 16-bit numbers and the crazy placement of bits within those, because in assembler you'll use human-readable abbreviations for that, so-called mnemonics, an aid to memory. The assembler representation for hex 9588 is simply the abbreviation "SLEEP". In contrast to 9588, SLEEP is easy to remember. Even for someone like me that has difficulties in remembering its own phone number.

Adding simply is "ADD". For naming the two registers, that are to be added, they are written as parameters. (No, not in brackets. C programmers, forget those brackets. You don't need those in assembler.) Simply type "ADD R0,R1". The line translates to a single 16 bit word, 0C01. The translation is done by the assembler.

The CPU only understands 0C01. The assembler translates the line to this 16 bit word, which is written to the flash storage, read from the CPU from there and executed. Each instruction that the CPU understands has such a mnemonic. And vice versa: each mnemonic has exactly one corresponding CPU instruction with a certain course of actions. The ability of the CPU determines the extent of instructions that are available in assembler. The language of the CPU is the base, the mnemonics only represent the abilities of the CPU itself.

2.4 Difference to high-level languages

Here some hints for high-level programmers. In high-level languages the constructions are not depending from the hardware or the abilities of a CPU. Those constructions work on very different processors, if there is a compiler for that language and for the processor family available. The compiler translates those language constructions to the processor's binary language. A GOTO in Basic looks like a JMP in assembler, but there is a difference in the whole concept between those two.

A transfer of program code to another processor hardware does only work if the hardware is able to do the same. If a processor CPU doesn't have access to a 16 bit timer, the compiler for a high-level language has to simulate one, using an 8-bit timer and some time-consuming code. If three timers are available, and the compiler is written for only two or a single timer, the available hardware remains unused. So you totally depend on the compiler's abilities, not on the CPU's abilities.

Another example with the above shown instruction "MUL". In assembler, the target processor determines if you can use this instruction or if you have to write a multiplication routine. If, in a high-level language, you use a multiplication the compiler inserts a math library that multiplies every kind of numbers, even if you have only 8-by-8-bit numbers and MUL alone would do it. The lib offers an integer, a long-word and some other routines for multiplications that you don't need. A whole package of things you don't really need. So you run out of flash in a small tiny AVR, and you change to a mega with 35 unused port pins. Or an xmega, just to get your elefant lib with superfluous routines into the flash. That is what you get from a simple "*", without even being asked.

2.5 Assembler is not machine language

Because assembler is closer to the hardware than any other language, it is often called machine language. This is not exact because the CPU only understands 16 bit instruction words in binary form. The string "ADD R0,R1" cannot be executed. And assembler is much simpler than machine language. Similarities between machine language and assembler are a feature, not a bug.

2.6 Interpreting and assembler

With an interpreter the CPU first translates the human-readable code into binary words that can be executed then. The interpreter would

* first read the text stream "A = A + B" (nine characters of one byte each),
* strip the four blanks from the text,
* locate the variables A and B (location in registers or in SRAM, precision/length, etc.),
* identify the plus sign as operator,
* prepare a machine executable sequence that is equivalent to the formulation in the text.

In the consequence, probably a simple machine code like "ADD R0,R1" (in Assembler) would result. But most probably the resulting machine code would be multiple words long (read and write variables from/to SRAM, 16-bit-integer adding, register saving/restoring on stack, etc., etc.).

The difference between the interpreter and the assembling is that, after assembling, the CPU gets its favored meal, executable words, directly. When interpreting the CPU is, during most of the time, performing the translation task. Translation probably requires 20 or 200 CPU steps, before the three or four words can be executed. Execution speed so is more than lame. While this is no problem if one uses a fast clock speed, it is inappropriate in time critical situations, where fast response to an event is required. No one knows what the CPU is just doing and how long this requires.

Not having to think about timing issues leads to the inability of the human programmer to resolve timing issues, and missing information on timing keeps him unable to do those things, if required.

2.7 High level languages and Assembler

High level languages insert additional nontransparent separation levels between the CPU and the source code. An example for such an nontransparent concept are variables. These variables are storages that can store a number, a text string or a single Boolean value. In the source code, a variable name represents a place where the variable is located, and, by declaring variables, the type (numbers and their format, strings and their length, etc.).

For learning assembler, just forget the high level language concept of variables. Assembler only knows bits, bytes, registers and SRAM bytes. The term "variable" has no meaning in assembler. Also, related terms like "type" are useless and do not make any sense here.

High level languages require you to declare variables prior to their first use in the source code, e. g. as Byte (8-bit), double word (16-bit), integer (15-bit plus 1 sign bit). Compilers for that language place such declared variables somewhere in the available storage space, including the 32 registers. If this placement is selected rather blind by the compiler or if there is some priority rule used, like the assembler programmer carefully does it, is depending more from the price of the compiler. The programmer can only try to understand what the compiler "thought" when he placed the variable. The power to decide has been given to the compiler. That "relieves" the programmer from the trouble of that decision, but makes him a slave of the compiler.

The instruction "A = A + B" is now type-proofed: if A is defined as a character and B a number (e. g. = 2), the formulation isn't accepted because character codes cannot be added with numbers. Programmers in high level languages believe that this type check prevents them from programming nonsense. The protection, that the compiler provides in this case by prohibiting your type error, is rather useless: adding 2 to the character "F" of course should yield a "H" as result, what else? Assembler allows you to do that, but not a compiler.

Assembler allows you to add numbers like 7 or 48 to add and subtract to every byte storage, no matter what type of thing is in the byte storage. What is in that storage, is a matter of decision by the programmer, not by a compiler. If an operation with that content makes sense is a matter of decision by the programmer, not by the compiler. If four registers represent a 32-bit-value or four ASCII characters, if those four bytes are placed low-to-high, high-to-low or completely mixed, is just up to the programmer. He is the master of placement, no one else. Types are unknown, all consists of bits and bytes somewhere in the available storage place. The programmer has the task of organizing, but also the chance of optimizing.

Of a similar effect are all the other rules, that the high level programmer is limited to. It is always claimed that it is saver and of a better overview to program anything in subroutines, to not jump around in the code, to hand over variables as parameters, and to give back results from functions. Forget most of those rules in assembler, they don't make much sense. Good assembler programming requires some rules, too, but very different ones. And, what's the best: most of them have to be created by yourself to help yourself. So: welcome in the world of freedom to do what you want, not what the compiler decides for you or what theoretical professors think would be good programming rules.

High level programmers are addicted to a number of concepts that stand in the way of learning assembler: separation in different access levels, in hardware, drivers and other interfaces. In assembler this separation is complete nonsense, separation would urge you to numerous workarounds, if you want to solve your problem in an optimal way.

Because most of the high level programming rules don't make sense, and because even puristic high level programmers break their own rules, whenever appropriate, see those rules as a nice cage, preventing you from being creative. Those questions don't play a role here. Everything is direct access, every storage is available at any time, nothing prevents your access to hardware, anything can be changed - and even can be corrupted. Responsibility remains by the programmer only, that has to use his brain to avoid conflicts when accessing hardware.

The other side of missing protection mechanisms is the freedom to do everything at any time. So, smash your ties away to start learning assembler. You will develop your own ties later on to prevent yourself from running into errors.

2.8 What is really easier in assembler?

All words and concepts that the assembler programmer needs is in the datasheet of the processor: the instruction and the port table. Done! With the words found there anything can be constructed. No other documents necessary. How the timer is started (is writing "Timer.Start(8)" somehow easier to understand than "LDI R16,0x02" and “OUT TCCR0,R16"?), how the timer is restarted at zero ("CLR R16” and “OUT TCCR0,R16"), it is all in the data sheet. No need to consult a more or less good documentation on how a compiler defines this or that. No special, compiler-designed words and concepts to be learned here, all is in the datasheet. If you want to use a certain timer in the processor for a certain purpose in a certain mode of the 15 different possible modes, nothing is in the way to access the timer, to stop and start it, etc.

What is in a high level language easier to write "A = A + B" instead of "MUL R16,R17"? Not much. If A and B aren't defined as bytes or if the processor type is tiny and doesn't understand MUL, the simple MUL has to be exchanged with some other source code, as designed by the assembler programmer or copy/pasted and adapted to the needs. No reason to import an nontransparent library instead, just because you're to lazy to start your brain and learn.

Assembler teaches you directly how the processor works. Because no compiler takes over your tasks, you are completely the master of the processor. The reward for doing this work, you are granted full access to everything. If you want, you can program a baud-rate of 45.45 bps on the UART. A speed setting that no Windows PC allows, because the operating system allows only multiples of 75 (Why? Because some historic mechanical teletype writers had those special mechanical gear boxes, allowing quick selection of either 75 or 300 bps.). If, in addition, you want 1 and a half stop bytes instead of either 1 or 2, why not programming your own serial device with assembler software. No reason to give things up.

Who is able to program in assembler has a feeling for what the processor allows. Who changes from assembler to a higher level language later on, e. g. in case of very complex tasks, has made the decision to select that on a rational basis. If someone skips learning assembler he has to do what he can, sticks to the available libraries and programs creative workarounds for things that the compiler doesn't allow, and in a way that assembler programmers would laugh at loud. The whole world of the processor is at the assembler programmer's command, so why do complicated and highly sensitive workarounds on something you can formulate in a nice, lean, esthetic way?

3 Hardware for AVR-Assembler-Programming

Learning assembler requires some simple hardware equipment to test your programs, and see if it works in practice.

This section shows two easy schematics that enable you to home brew the required hardware and gives you the necessary hints on the required background. This hardware really is easy to build. I know nothing easier than that to test your first software steps. If you like to make more experiments, leave some more space for future extensions on your experimental board.

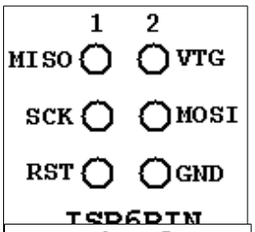
If you don't like the smell of soldering, you can buy a ready-to-use board, too. The available boards are characterized in this section below.

3.1 The ISP Interface of the AVR processor family

Before going into practice, we have to learn a few essentials on the serial programming mode of the AVR family. No, you don't need three different voltages to program and read an AVR flash memory. No, you don't need another pre-programmed microprocessor to program the AVRs. No, you don't need 10 I/O lines to tell the chip what you like it to do. And you don't even have to remove the AVR from the socket on your your experimental board, before programming it. It's even easier than that.

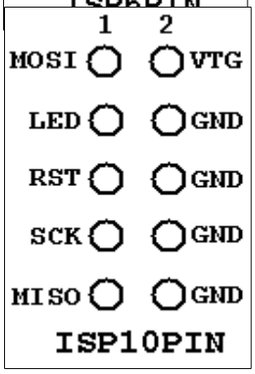
All this is done by a build-in interface in the AVR chips, that enable you to write and read the content of the program flash and the built-in-EEPROM. This interface works serially and needs only three signal lines:

- SCK: A clock signal that shifts the bits to be written to the memory into an internal shift register, and that shifts out the bits to be read from another internal shift register,
- MOSI: The data signal that sends the bits to be written to the AVR,
- MISO: The data signal that receives the bits read from the AVR.



These three signal pins are internally connected to the programming machine only if you change the RESET (sometimes also called RST or restart) pin to zero. Otherwise, during normal operation of the AVR, these pins are programmable I/O lines like all the others.

If you like to use these pins for other purposes during normal operation, and for in-system-programming, you'll have to take care, that these two purposes do not conflict. Usually you then decouple these by resistors or by use of a multiplexer. What is necessary in your case, depends from your use of the pins in the normal operation mode. You're lucky, if you can use them for in-system-programming exclusively.



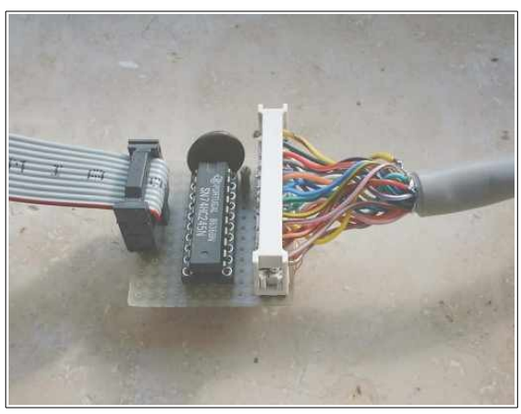
Not necessary, but recommendable for in-system-programming is, that you supply the programming hardware out of the supply voltage of your system. That makes it easy, and requires two additional lines between the programmer and the AVR board. GND is the common ground or negative pole of the supply voltage, VTG (target voltage) the supply voltage (usually +5.0 volts). This adds up to 6 lines between the programmer hardware and the AVR board. The resulting ISP6 connection, as defined by ATMEL, is shown on the left.

Standards always have alternative standards, that were used earlier. This is the technical basis that constitutes the adapter industry. In our case the alternative standard was designed as ISP10 and was used on the STK200 board, sometimes also called CANDA interface. It's still a very widespread standard, and even the more recent STK500 board is equipped with it. ISP10 has an additional signal to drive a red LED. This LED signals that the programmer is doing his job. A good idea. Just connect the LED to a resistor and clamp it the positive supply voltage.

3.2 Programmer for the PC-Parallel-Port

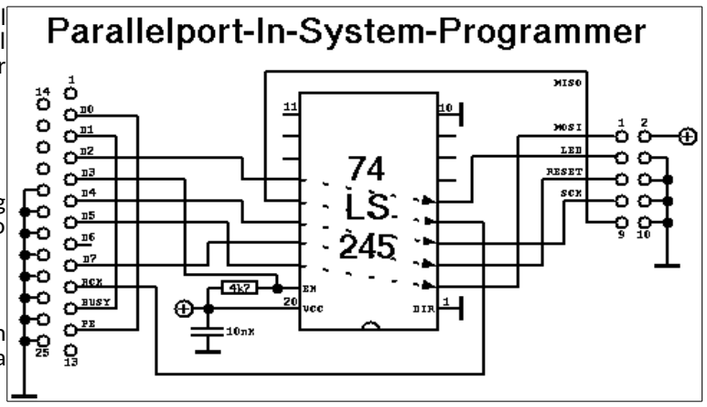
Now, heat up your soldering iron and build up your programmer. It is a quite easy schematic and works with standard parts from your well-sorted experiments box.

Yes, that's all you need to program an AVR. The 25-pin plug goes into the parallel port of your PC, the 10-pin ISP goes to your AVR experimental board. If your box doesn't have a 74LS245, you can also use a 74HC245 (with no hardware changes) or a 74LS244/74HC244 (by changing some pins and signals). If you use HC, don't forget to tie unused inputs either to GND or the supply voltage, otherwise the buffers might produce extra noise by capacitive switching.



The necessary program algorithm is done by the ISP software. Be aware that this parallel port interface is not supported by ATMEL's studio software any more. So, if you want to program your AVR directly from within the studio, use different programmers. The Internet provides several solutions.

If you already have a programming board, you will not need to build this programmer, because you'll find the ISP interface on some pins. Consult your handbook to locate these.



3.3 Experimental boards

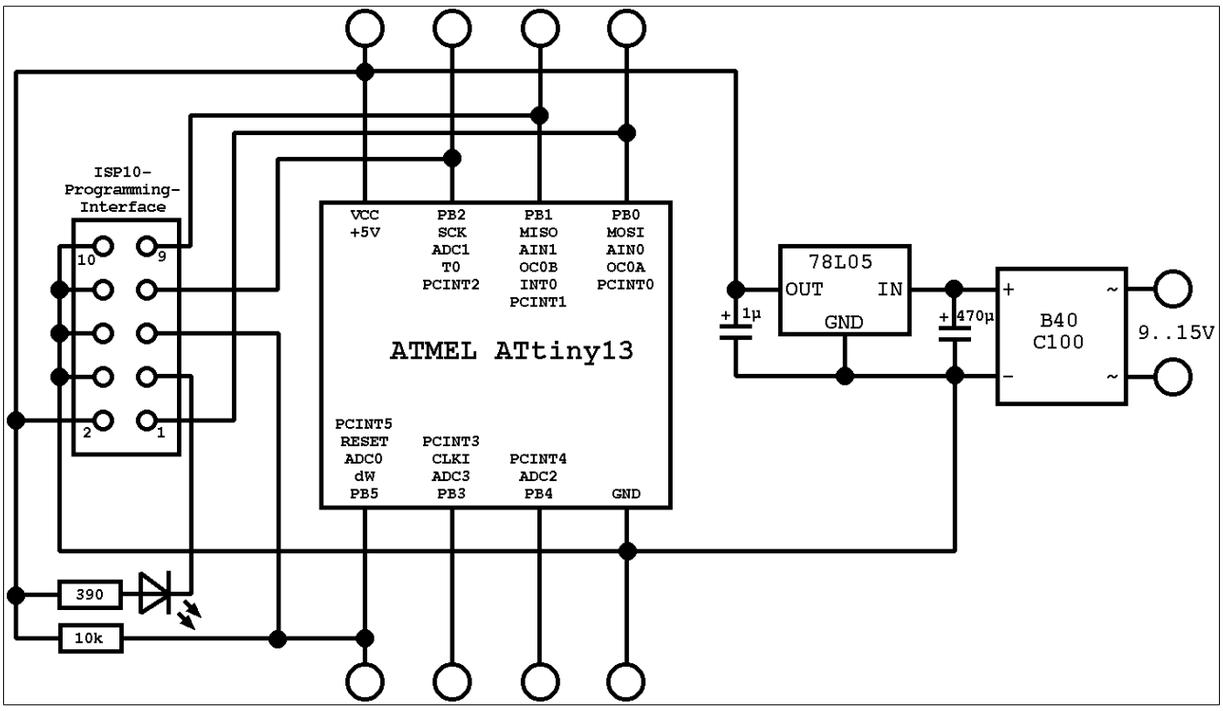
You probably want to do your first programming steps with a self-made AVR board. Here are two versions offered:

- A very small one with an ATtiny13, or
- a more complicated one with an AT90S2313 or ATmega2313, including a serial RS232 interface.

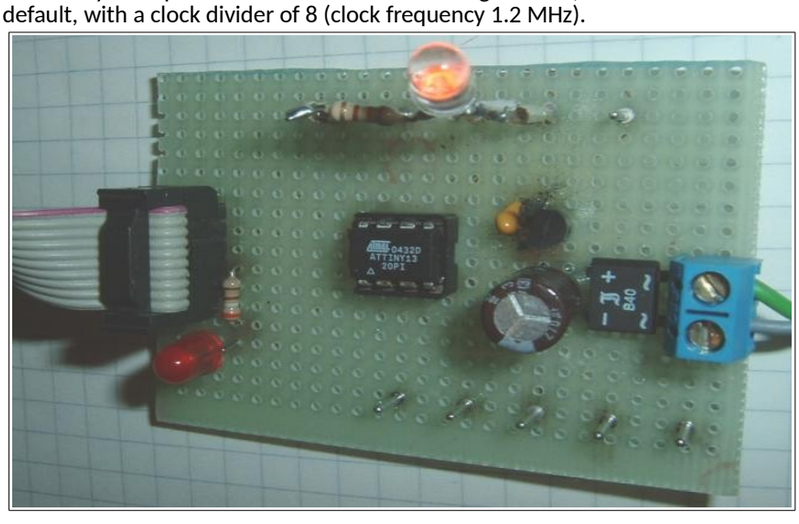
3.3.1 Experimental board with an ATtiny13

This is a very small board that allows experiments with the ATtiny13's internal hardware. The picture shows

- the ISP10 programming interface on the left, with a programming LED attached via a resistor of 390 Ohms,
- the ATtiny13 with a pull-up of 10k on its RESET pin (pin 1),
- the supply part with a bridge rectifier, to be supplied with 9..15V from an AC or DC source, and a small 5V regulator.



The ATtiny13 requires no external XTAL or clock generator, because it works with its internal 9.6 MHz RC generator and, by default, with a clock divider of 8 (clock frequency 1.2 MHz).



The hardware can be build on a small board like the one shown in the picture. All pins of the tiny13 are accessible, and external hardware components, like the LED shown, can be easily plugged in.

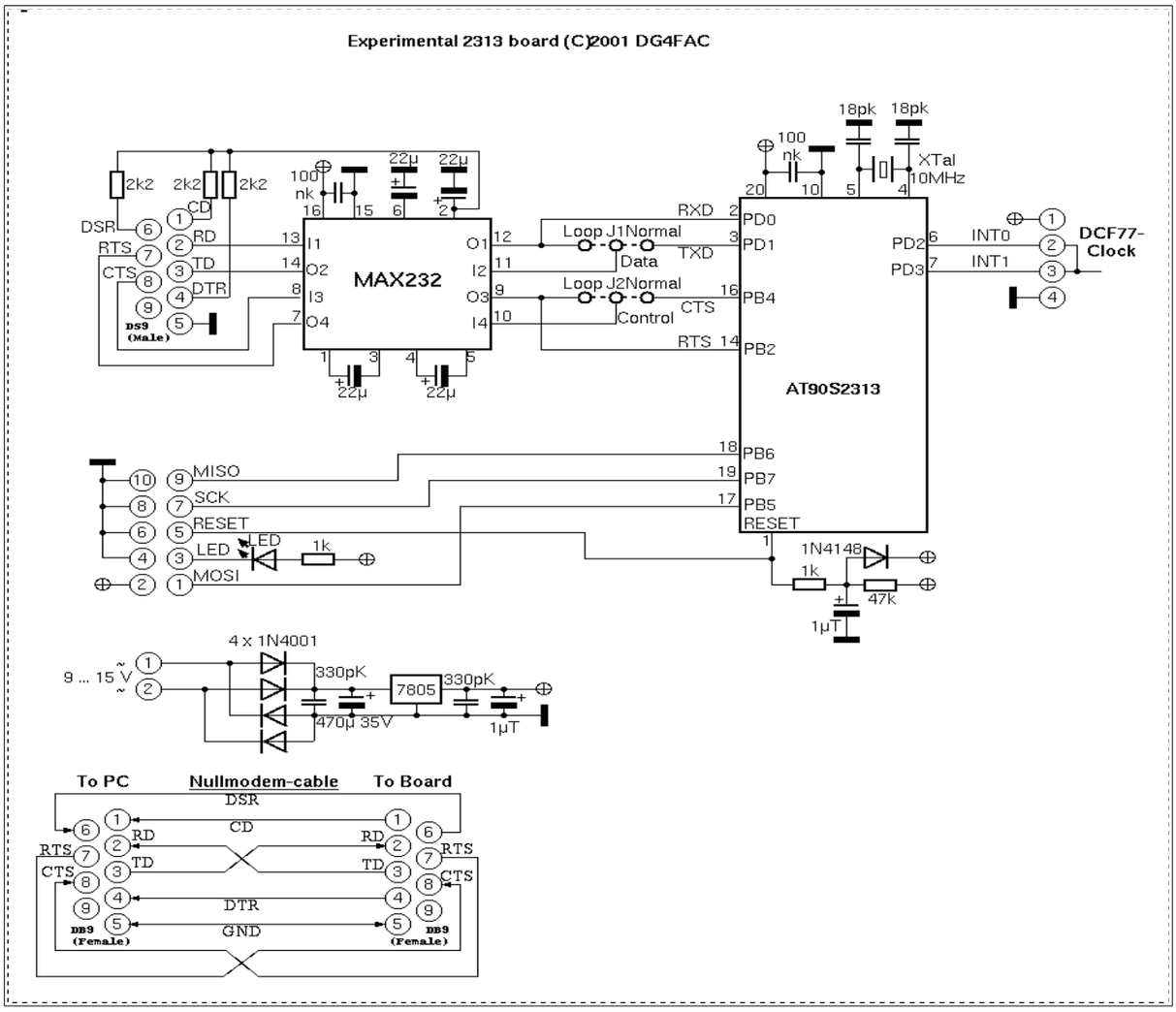
This board allows the use of the ATtiny13's hardware components like I/O-ports, timers, AD converters, etc.

3.3.2 Experimental board with an AT90S2313/ATmega2313

For test purposes, were more I/O-pins or a serial communication interface is necessary, we can use a AT90S2313 or ATmega2313 on an experimental board. The schematic shows

- a small voltage supply for connection to an AC transformer and a voltage regulator 5V/1A,
- a XTAL clock generator (here with a 10 MHz XTAL, all other frequencies below the maximum for the 2313 will also work),
- the necessary parts for a safe reset during supply voltage switching,
- the ISP-Programming-Interface (here with a ISP10PIN-connector).

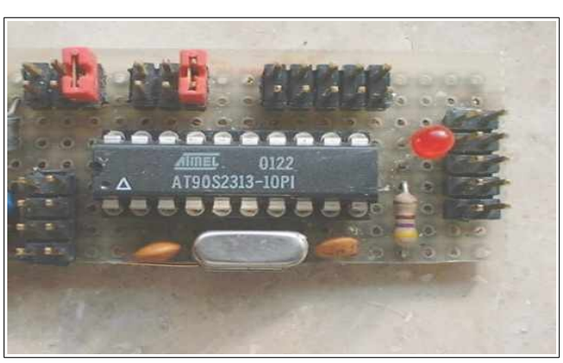
So that's what you need to start with. Connect other peripheral add-ons to the numerous free I/O pins of the 2313.



The easiest output device can be a LED, connected via a resistor to the positive supply voltage. With that, you can start writing your first assembler program switching the LED on and off.

If you

- do not need the serial communication interface, just skip the hardware connected to pins 2/3 and 14/16,
- do not need hardware handshake signals, skip the hardware on the pins 14/16 and connect RTS on the 9-pin-connector over a 2.2k resistor to +9V.



If you use an ATmega2313 instead of an AT90S2313, the following changes are resulting:

- the external XTAL is not necessary, as the ATmega has an internal RC clock generator, so just skip all connections to pins 4 and 5,
- if you want to use the external XTAL instead of the build-in RC as clock source, you will have to program the fuses of the ATmega accordingly.

3.4 Ready-to-use commercial programming boards for the AVR-family

If you do not like home-brewed hardware, and if have some extra money left that you don't know what to do with, you can buy a commercial programming board. Depending from the amount of extra money you'd like to spend, you can select

between more or less costly versions. For the amateur the following selection criteria should be looked at:

- price,
- PC interface (preferably USB, less convenient or durable: 9-pin RS232, requiring additional software: interfaces for the parallel port of the PC),
- support reliability for newer devices (updates are required from time to time, otherwise you sit on a nearly dead horse),
- hardware features (depends on your foreseeable requirements in the next five years).

The following section describes the three standard boards of ATMEL, the STK200, the STK500 and the Dragon. The selection is based on my own experiences and is not a recommendation.

3.4.1 STK200

The STK200 from ATMEL is a historic board. If you grab a used one you'll get

- a board with some sockets (for 8, 20, 28 and 40 pin devices),
- eight keys and LEDs, hard connected to port D and B,
- an LCD standard 14-pin interface,
- an option for attaching a 28-pin SRAM,
- a RS232 interface for communication,
- a cable interface for a PC parallel port on one side and a 10-pin ISP on the other side.

High voltage programming is not supported.

The board cannot be programmed from within the Studio, the programming software is no longer maintained, and you must use external programs capable of driving the PC parallel port.

If someone offers you such a board, take it only for free and if you're used to operate software of the necessary kind.

3.4.2 STK500

Easy to get is the STK500 (e. g. from ATMEL). It has the following hardware:

- Sockets for programming most of the AVR types (e. g. 14-pin devices or TQFP packages require additional hardware),
- serial and parallel programming in normal mode or with high voltage (HV programming brings devices back to life even if their RESET pin has been fuse-programmed to be normal port input),
- ISP6PIN- and ISP10PIN-connection for external In-System-Programming,
- programmable oscillator frequency and supply voltages,
- plug-in switches and LEDs,
- a plugged RS232C-connector (UART),
- a serial Flash-EEPROM (only older boards have this),
- access to all port pins via 10-pin connectors.

A major disadvantage of the board is that, before programming a device, several connections have to be made manually with the delivered cables.

The board is connected to the PC using a serial port (COMx). If your laptop doesn't have a serial interface, you can use one of the common USB-to-Serial-Interface cables with a software driver. In that case the driver must be adjusted to use between COM1 and COM8 and a baud rate of 115k to be automatically detected by the Studio software.

Programming is performed and controlled by recent versions of AVR studio, which is available for free from ATMEL's web page after registration. Updates of the device list and programming algorithm are provided with the Studio versions, so the support for newer devices is more likely than with other boards and programming software.

Experiments can start with the also supplied AVR (older versions: AT90S8515, newer boards versions include different types). This covers all hardware requirements that the beginner might have.

3.4.3 AVR Dragon

The AVR dragon is a very small board. It has an USB interface, which also supplies the board and the 6-pin-ISP interface. The 6-pin-ISP-Interface is accompanied by a 20-pin HV programming interface. The board is prepared for adding some sockets on board, but doesn't have sockets for target devices and other hardware on board.

The dragon is supported by the Studio software and is a updated automatically.

Its price and design makes it a nice gift for an AVR amateur. The box fits nicely in a row with other precious and carefully designed boxes.

4 Tools for AVR assembly programming

Four basic programs are necessary for assembly programming. These tools are:

- the editor,
- the assembler program,
- the chip programing interface, and
- the simulator.

Two different basic routes are possible:

1. anything necessary in one package,
2. each task is performed with a specific program, the results are stored as specific files.

Usually route #1 is chosen. But because this is a tutorial, and you are to understand the underlying mechanism first, we start with the description of route #2 first. This shows the way from a text file to instruction words in the flash memory

4.1 The editor

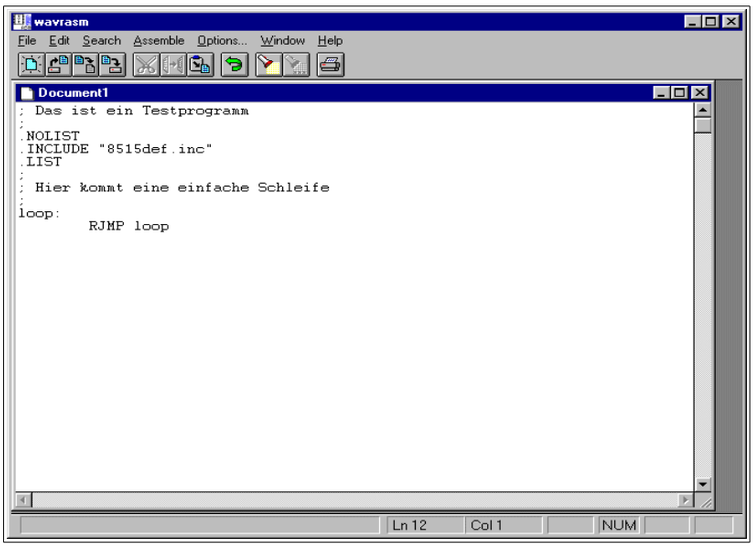
4.1.1 A simple typewriter

Assembler programs are written with an editor. The editor just has to be able to create and edit ASCII text files. So, basically, any simple editor does it.

Some features of the editor can have positive effects:

- Errors, that the assembler later detects, are reported along with the line number in the text file. Line numbers are also a powerful invention of the computer-age when it comes to discussions on your code with someone else.
- Typing errors are largely reduced, if those errors are marked with colors. It is a nice feature of an editor to highlight the components of a line in different colors. More or less intelligent recognition of errors ease typing. But this is a feature that I don't really miss.
- If your editor allows the selection of fonts, chose a font with fixed spacing, like Courier. Headers look nicer with that.
- Your editor should be capable of recognizing line ends with any combination of characters (carriage returns, line feeds, both) without producing unacceptable screens. Another item on the wish list for Widows 2013.

If you prefer shooting with cannons to kill sparrows, you can use a mighty word processing software to write assembler programs. It might look nicer, with large bold headings, gray comments, red warnings, changes marked, and reminders on To-Do's in extra bubble fields. Some disadvantages here: you have to convert your text to plain text at the end, losing all your nice design work, and your resulting text file should not have a single control byte left. Otherwise this single byte will cause an error message, when you assemble the text. And remember: Line numbers here are only correct on page one of your source code.



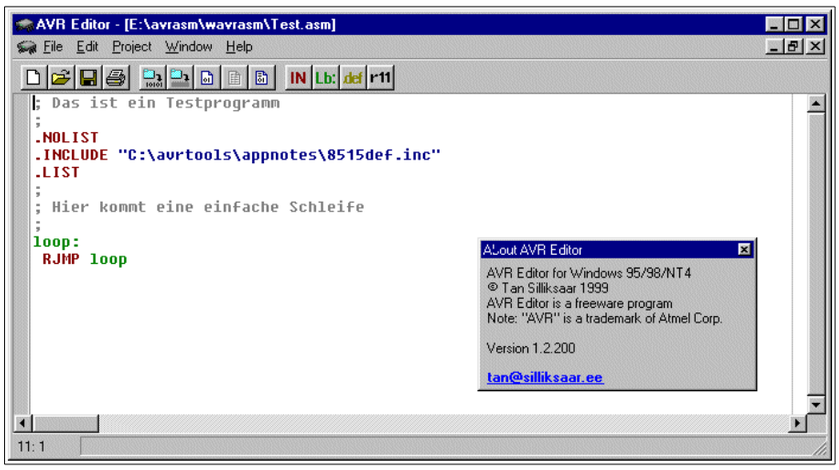
So, whatever text program you chose, it's up to you. The following examples are written in wavrasm, an editor provided by ATMEL in earlier days.

In the plain editor field we type in our directives and assembly instructions. It is highly recommended that lines come together with some comments (starting with ;). Later understanding of what we've planned here will be helpful in later debugging.

Now store the program text, named to something.asm into a dedicated directory, using the file menu. The assembly program is complete now.

If you'd like to see what syntax-highlighting means, I have a snapshot of such an AVR editor here.

The editor recognizes instructions automatically and uses different colors (syntax highlighting) to signal user constants and typing errors in those instructions (in black). Storing the code in an .asm file provides nearly the same text file, colors are not stored in the file.



Don't try to find this editor or its author; the editor is history and no longer maintained.

4.1.2 Structuring assembler code

This page shows the basic structure of an assembler program. These structures are typical for AVR assembler. This text discusses

- comments,
- header informations,
- code at program start and
- the general structure of programs.

Comments

The most helpful things in assembler programs are comments. If you need to understand older code that you wrote, sometimes years after, you will be happy about having some or more hints what is going on in that line. If you like to keep your ideas secret, and to hide them against yourself and others: don't use comments. A comment starts with a semicolon. All that follows behind on the same line will be ignored by the compiler. If you need to write a comment over multiple lines, start each line with a semicolon. So each assembler program should start like that:

```
;
; Click.asm, Program to switch a relais on and off each two seconds
; Written by G.Schmidt, last change: 7.10.2001
;
```

Put comments around all parts of the program, be it a complete subroutine or a table. Within the comment mention the special nature of the routine, pre-conditions necessary to call or run the routine. Also mention the results of the subroutine in case you later will have to find errors or to extend the routine later. Single line comments are defined by adding a semicolon behind the command on the line. Like this:

```
LDI R16,0x0A ; Here something is loaded
MOV R17,R16 ; and copied somewhere else
```

Things to be written on top

Purpose and function of the program, the author, version information and other comments on top of the program should be followed by the processor type that the program is written for, and by relevant constants and by a list with the register names. The processor type is especially important. Programs do not run on other chip types without changes. The instructions are not completely understood by all types, each type has typical amounts of EEPROM and internal SRAM. All these special features are included in a header file that is named xxxxdef.inc, with xxxx being the chip type, e. g. 2313, tn2323, or m8515. These files are available by ATMEL. It is good style to include this file at the beginning of each program. This is done like that:

```
.NOLIST ; Don't list the following in the list file
.INCLUDE "m8515def.inc" ; Import of the file
.LIST ; Switch list on again
```

The path, where this file can be found, is only necessary if you don't work with ATMEL's Studio. Of course you have to include the correct path to fit to your place where these files are located. During assembling, the output of a list file listing the results is switched on by default. Having listing ob might result in very long list file (*.lst) if you include the header file. The directive .NOLIST turns off this listing for a while, LIST turns it on again. Let's have a short look at the header file. First these files define the processor type:

```
.DEVICE ATMEGA8515 ; The target device type
```

The directive .DEVICE advices the assembler to check all instructions if these are available for that AVR type. It results in an error message, if you use code sequences that are not defined for this type of processor. You don't need to define this within your program as this is already defined within the header file. The header file also defines the registers XH, XL, YH, YL, ZH and ZL. These are needed if you use the 16-bit-pointers X, Y or Z to access the higher or lower byte of the pointer separately. All port locations are also defined in the header file, so PORTB translates to a hex number where this port is located on the defined device. The port's names are defined with the same names that are used in the data sheets for the respective processor type. This also applies to single bits in the ports. Read access to port B, Bit 3, can be done using its bit name PINB3, as defined in the data sheet. In other words: if you forget to include the header file you will run into a lot of error messages during assembly. The resulting error messages are in some cases not necessarily related to the missing header file. Others things that should be on top of your programs are the register definitions you work with, e. g.:

```
.DEF mpr = R16 ; Define a new name for register R16
```

This has the advantage of having a complete list of registers, and to see which registers are still available and unused. Renaming registers avoids conflicts in the use of these registers and the names are easier to remember. Further on we define the constants on top of the source file, especially those that have a relevant role in different parts of the program. Such a constant would, e. g., be the Xtal frequency that the program is adjusted for, if you use the serial interface on board. With

```
.EQU fq = 4000000 ; XTal frequency definition
```

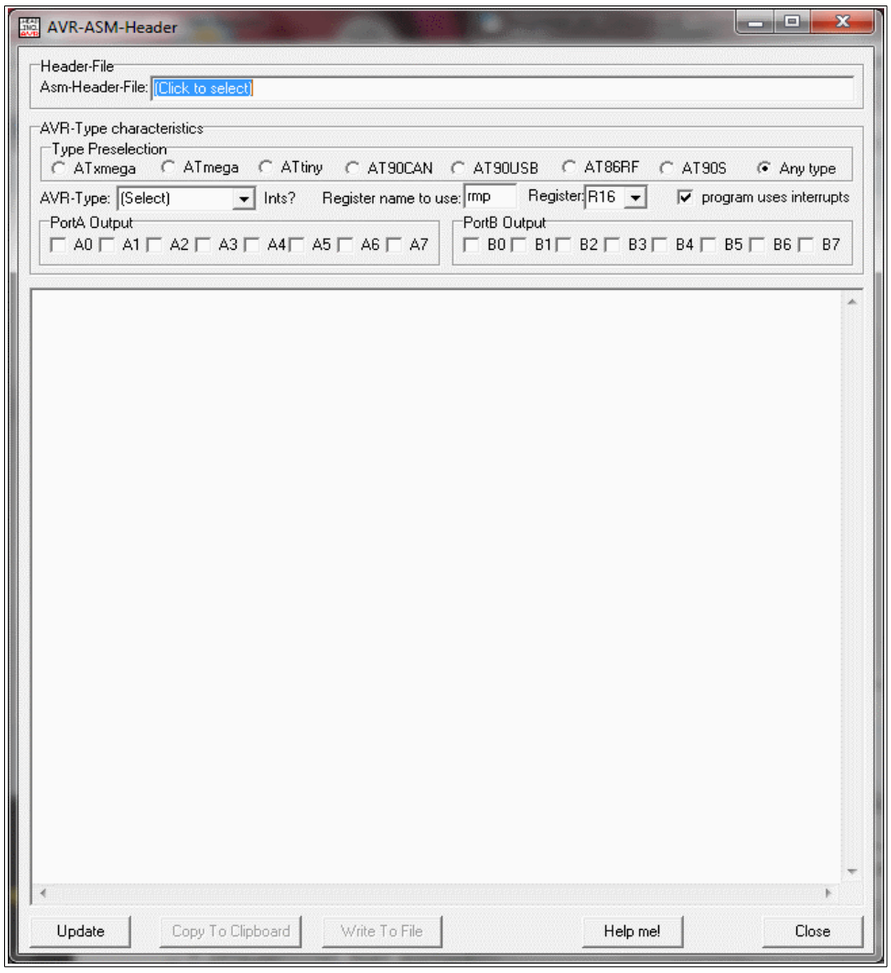
at the beginning of the source code you immediately see for which clock you wrote the program. Very much easier than searching for this information within 1482 lines of source code.

Things that should be done at program start

After you have done the header, the program code should start. At the beginning of the code the reset- and interrupt-vectors (their function see in the JUMP section) are placed. As these require relative jumps, we should place the respective interrupt service routines right behind. In case of ATmega types with larger flash memory JUMP instructions can be used here, so be careful here. There is some space left then for other subroutines, before we place the main program. The main program always starts with initialization of the stack pointer, setting registers to default values, and the init of the hardware components used. The following code is specific for the program.

Tool for structuring of program code

The described standardized structure is included in a program written for Windows Operating Systems, which can be downloaded at http://www.avr-asm-download.de/avr_head.zip.



Unzip the executable file, and simply run it. It shows this: Here you can choose **ATtiny** by clicking on it, and then select **ATtiny13** in the dropdown field **AVR-Type**.

You are now asked to navigate to its respective include-file tn13def.inc. Show the program the way where the header file is located.
Here you can enter your desired multi purpose register, the output configuration on ports A and B, if available, and if you want to use interrupts.

Click **Update** to fill the window with your code frame.

Click **CopyToClipboard**, if you want to paste this code into your code editor, or **WriteToFile** to write this to an assembler code file instead.

If you don't know what it is for and what to do, press the **Help** button.
This produces the following code:

```
;
;
*****************************************
; * [Add Project title here]              *
; * [Add more info on software version here] *
; * (C)20xx by [Add Copyright Info here]     *
; *****************************************
;
; Included header file for target AVR type
.NOLIST
.INCLUDE "tn13def.inc" ; Header for ATTINY13
.LIST
;
; =============================================
;   H A R D W A R E   I N F O R M A T I O N
; =============================================
;
; [Add all hardware information here]
;
; =============================================
;      P O R T S   A N D   P I N S
; =============================================
```

```
;
; [Add names for hardware ports and pins here]
; Format: .EQU Controlportout = PORTA
;         .EQU Controlportin = PINA
;         .EQU LedOutputPin = PORTA2
;
; =============================================
;    C O N S T A N T S   T O   C H A N G E
; =============================================
;
; [Add all constants here that can be subject
;  to change by the user]
; Format: .EQU const = $ABCD
;
; =============================================
;  F I X + D E R I V E D   C O N S T A N T S
; =============================================
;
; [Add all constants here that are not subject
;  to change or calculated from constants]
; Format: .EQU const = $ABCD
;
; =============================================
;   R E G I S T E R   D E F I N I T I O N S
; =============================================
;
; [Add all register names here, include info on
;  all used registers without specific names]
; Format: .DEF rmp = R16
.DEF rmp = R16 ; Multipurpose register
;
; =============================================
;       S R A M   D E F I N I T I O N S
; =============================================
;
.DSEG
.ORG  0X0060
; Format: Label: .BYTE N ; reserve N Bytes from Label:
;
; =============================================
;   R E S E T   A N D   I N T   V E C T O R S
; =============================================
;
.CSEG
.ORG $0000
        rjmp Main ; Reset vector
        reti ; Int vector 1
        reti ; Int vector 2
        reti ; Int vector 3
        reti ; Int vector 4
        reti ; Int vector 5
        reti ; Int vector 6
        reti ; Int vector 7
        reti ; Int vector 8
        reti ; Int vector 9
;
; =============================================
;     I N T E R R U P T   S E R V I C E S
; =============================================
;
; [Add all interrupt service routines here]
;
; =============================================
;     M A I N    P R O G R A M    I N I T
; =============================================
;
Main:
; Init stack
        ldi rmp, LOW(RAMEND) ; Init LSB stack
        out SPL,rmp
; Init Port B
        ldi rmp,(1<<DDB2)|(1<<DDB1)|(1<<DDB0) ; Direction of Port B
        out DDRB,rmp
; [Add all other init routines here]
        ldi rmp,1<<SE ; enable sleep
        out MCUCR,rmp
```

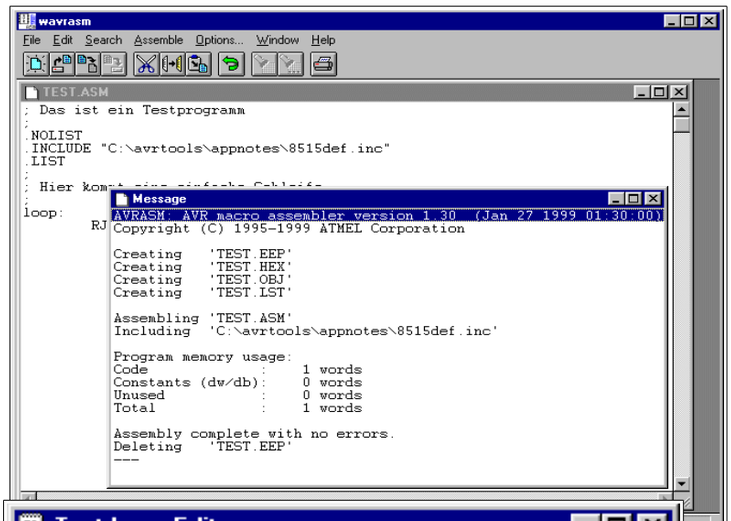
```
        sei
;
; =============================================
;         P R O G R A M    L O O P
; =============================================
;
Loop:
        sleep ; go to sleep
        nop ; dummy for wake up
        rjmp loop ; go back to loop
;
; End of source code
;
```

4.2 The assembler

Now we have a text file, with blank ASCII characters. The next step is to translate this code to a machine-oriented form well understood by the AVR chip. Doing this is called assembling, which means „put together the right instruction words“. The program that reads the text file and produces some kind of output files is called Assembler. In the easiest form this is a instruction-line program that, when called, expects the address of the text file and some optional switches, and then starts assembling the instructions found in the text file.

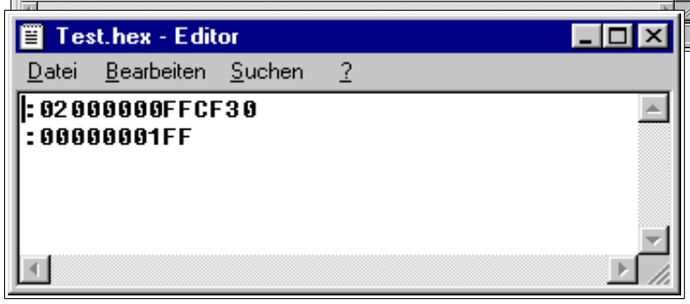
If your editor allows calling external programs, this is an easy task. If not (another item on the wish list for the editor in Widows 2010), it is more convenient to write a short batch file (again using an editor). That batch file should have a line like this:

PathToAssembler\Assembler.exe -options PathToTextfile\Textfile.asm



Clicking on the editor's external program caller or on the batch file starts the command line assembler. That piece of software reports the complete translation process (in the smaller window), here with no errors. If errors occur these are notified, along with their type and line number. Assembling resulted in one word of code which resulted from the RJMP instruction that we used. Assembling our single asm text file now has produced four other files (not all apply here).

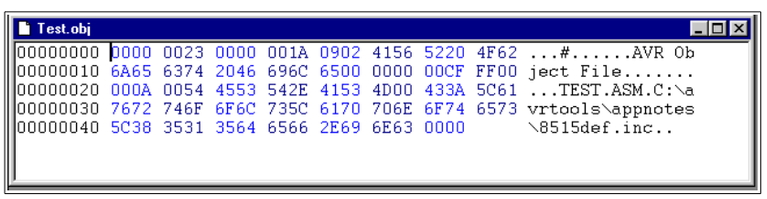
The first of these four new files, TEST.EEP, holds the content that should be written to the EEPROM of the AVR. This is not very interesting in our case, because we didn't program any content for the EEPROM. The assembler has therefore deleted this file when he completed the assembly run, because it is empty.

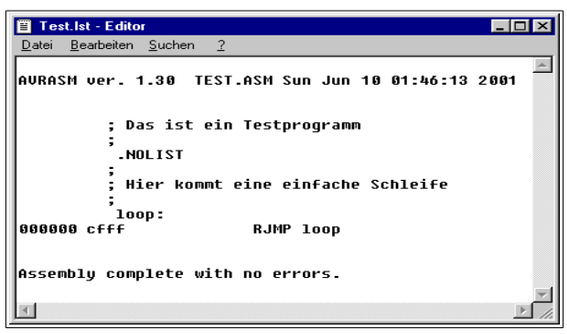


The second file, TEST.HEX, is more relevant because this file holds the instructions later programmed into the AVR chip. This file looks like this.

The hex numbers are written in a special ASCII form, together with address information and a check-sum for each line. This form is called Intel-hex-format, and is very old and stems from the early world of computing. The form is well understood by the programing software.

The third file, TEST.OBJ, will be introduced later, this file is needed to simulate an AVR. Its format is hexadecimal and defined by ATMEL. Using a hex-editor its content looks like this. Attention: This file format is not compatible with the programmer software, don't use this file to program the AVR (a very common error when starting). OBJ files are only produced by certain ATMEL assemblers, don't expect these files with other assemblers.





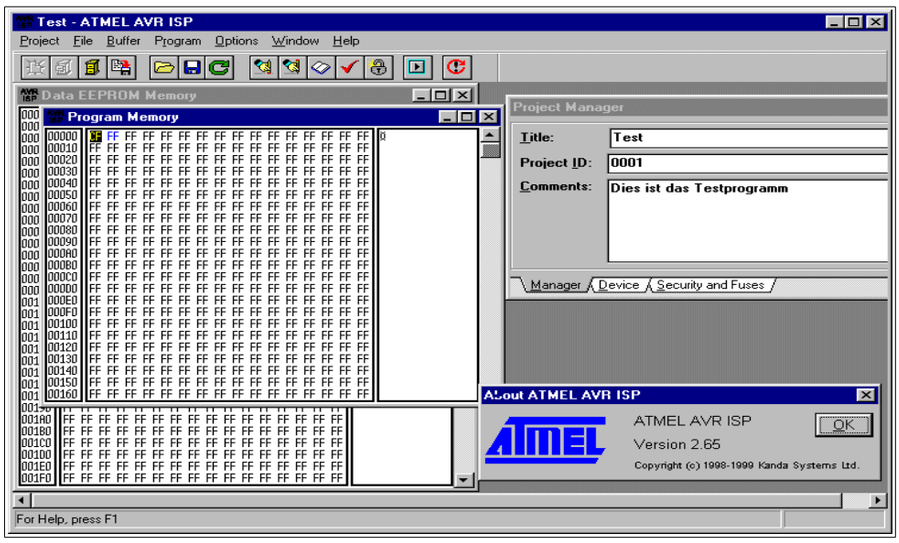
The fourth file, TEST.LST, is a text file. Display its content with a simple editor. The following results.

The program with all its addresses, instructions and error messages are displayed in a readable form. You will need that file in some cases to debug errors.

Listfiles are generated only if the appropriate option is selected on the command line options and if the .NOLIST directive doesn't suppress listing.

4.3 Programming the chips

To program our hex code, as coded in text form in the .HEX-file, to the AVR a programmer software is necessary. This software reads the .HEX-file and transfers its content, either bit-by-bit (serial programming) or byte-by-byte (parallel programming) to the AVR's flash memory. We start the programmer software and load the hex file that we just generated.



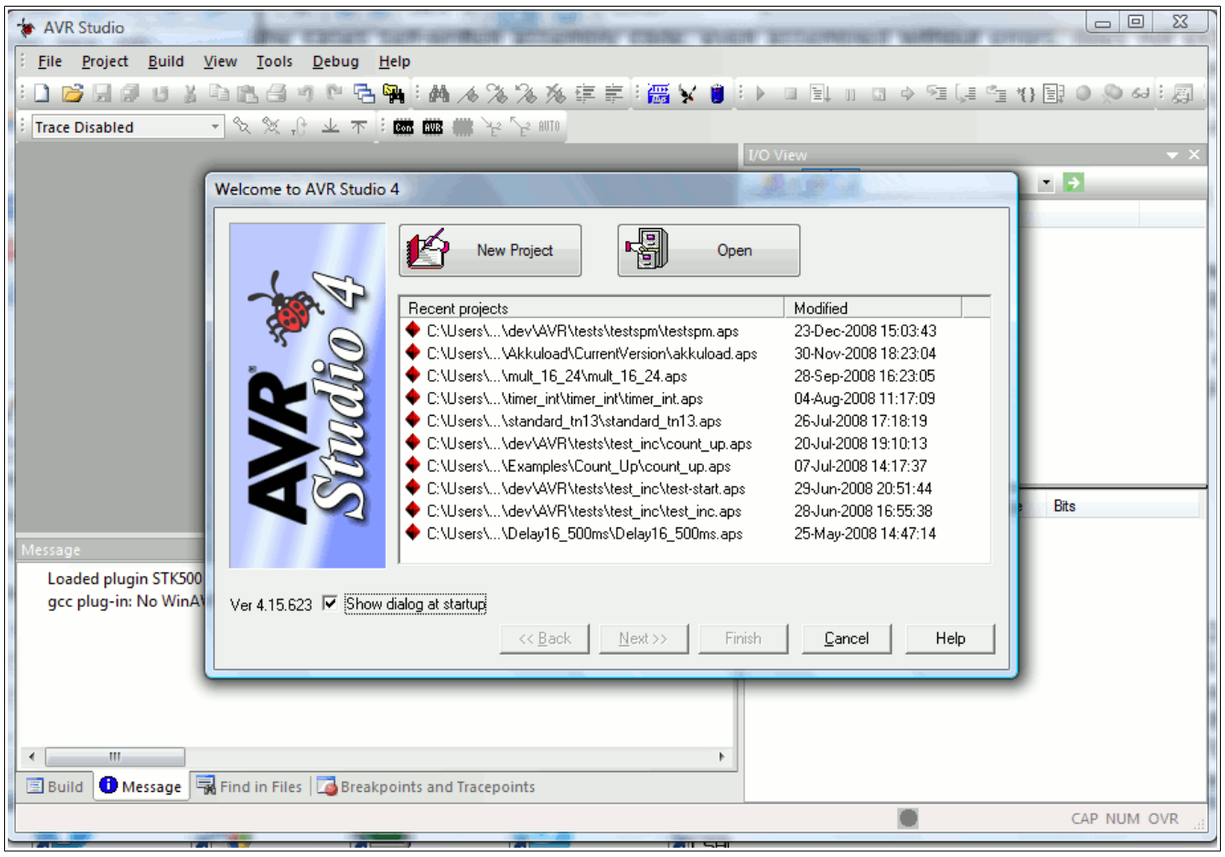
In an example that looks like this. Please note: the displayed window stems from ISP.exe, a historic program no longer distributed by ATMEL. Other programmer software looks similar.

The software will burn our code in the chip's program store. There are a number of preconditions necessary for this step and several reasons possible, if this step fails. Consult your programmer software help, if problems occur.

Programming hardware and appropriate software alternatives for different PC operating systems are available on the Internet. As an example for programming over the PC's parallel or serial communication port, PonyProg2000 should be mentioned here.

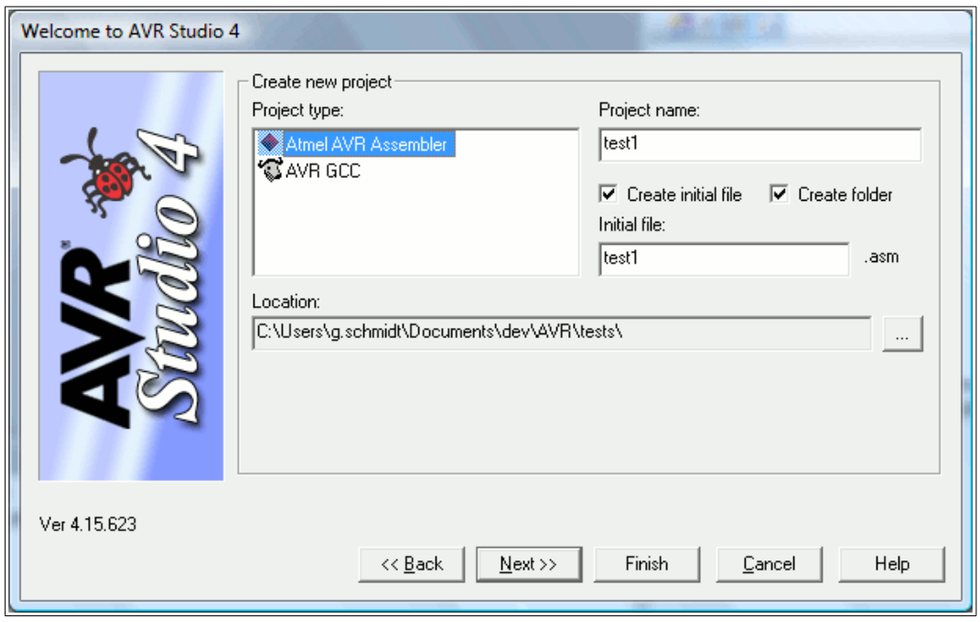
4.4 Simulation in the studio

In some cases self-written assembly code, even assembled without errors, does not exactly do what it should do when burned into the chip. Testing the software on the chip could be complicated, esp. if you have a minimum hardware and no opportunity to display interim results or debugging signals. In these cases the Studio software package from ATMEL provides ideal opportunities for debugging. Testing the software or parts of it is possible, the program code could be tested step-by-step displaying results.



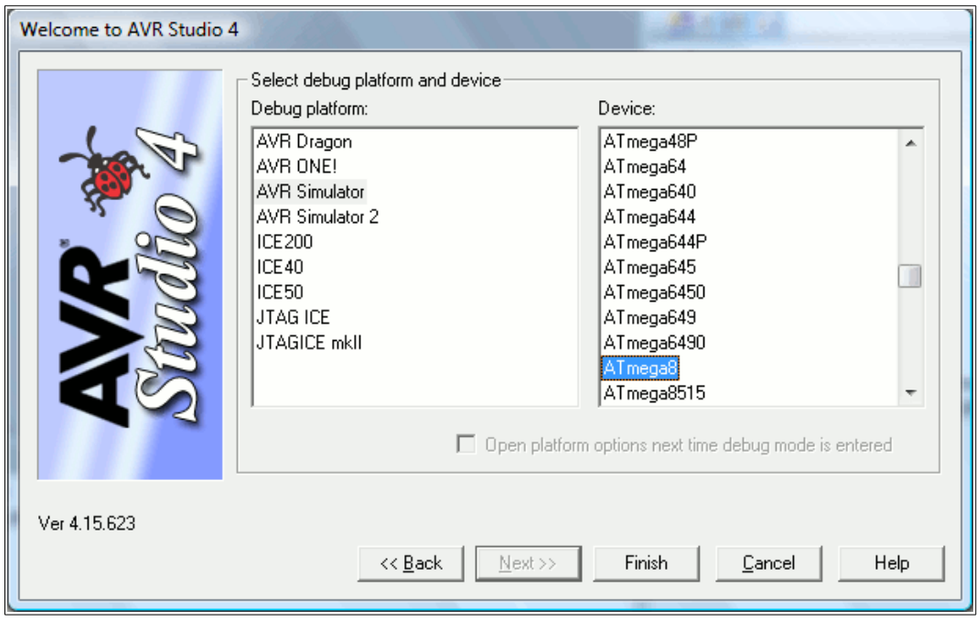
The pictures shown here are taken from Version 4 of the Studio, that is available for free on ATMEL's website. Older versions looks different, but do nearly the same. The Studio is a software that has all you need to develop, debug, simulate and burn your assembler programs into the AVR type of your choice. The studio is started and looks like this.

The first dialog asks whether an existing project should be opened or a new project is to be started. In case of a newly installed Studio “New Project” is the correct answer. The Button “Next>>” brings you to the settings dialog of your new project.

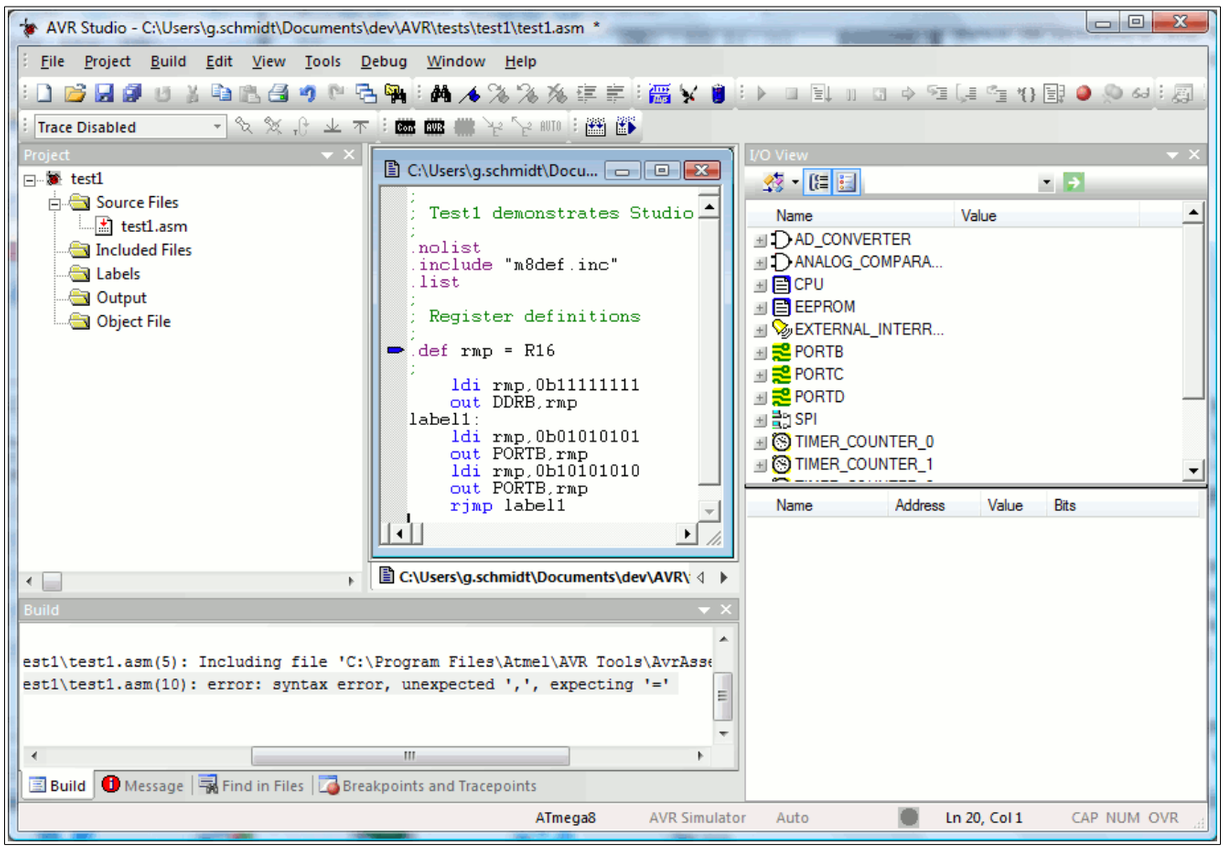


Here you select “Atmel AVR Assembler” as your project type, give that project a name (here “test1”) and let the Studio crate an initial (empty) file for your source code, let it create a folder and select a location for that project, where you have write access to.

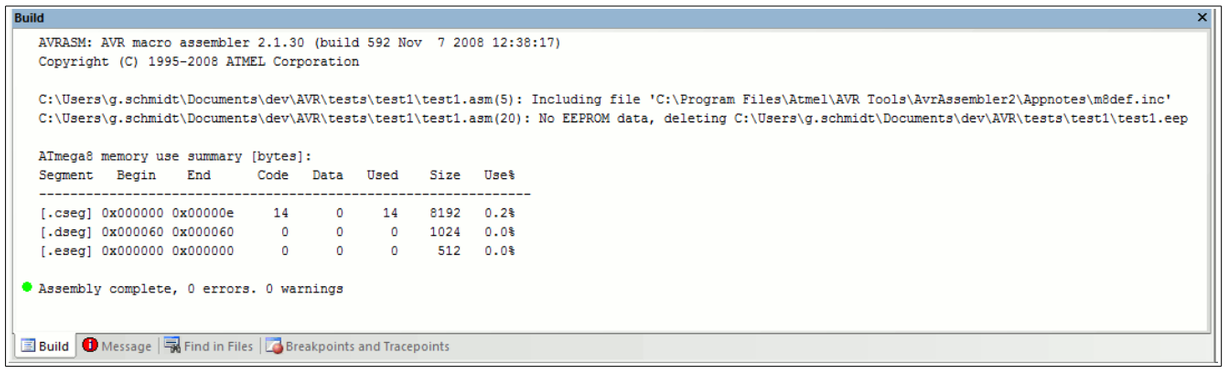
The button “Next>>” opens the platform and device selection dialog:



As debug platform select either “AVR simulator” or “AVR simulator 2”. As Device select your AVR type, here an ATmega8 was selected. If your desired type is grayed out, select another simulator platform. Close this window with the “Finish” button. Now a large window pops up, which has lots of different sub-windows.



On the left, the project window allows you to manipulate and view all your project files. In the middle, the editor window, allows you to write your source code (try typing its content to your editor window, don't care about the colors – these are added by the editor – remember syntax-highlighting?). On the left bottom is a “Build” section, where all your error messages go to. On the right side is a strange I/O view and below a rather white field, we'll come to that later on.



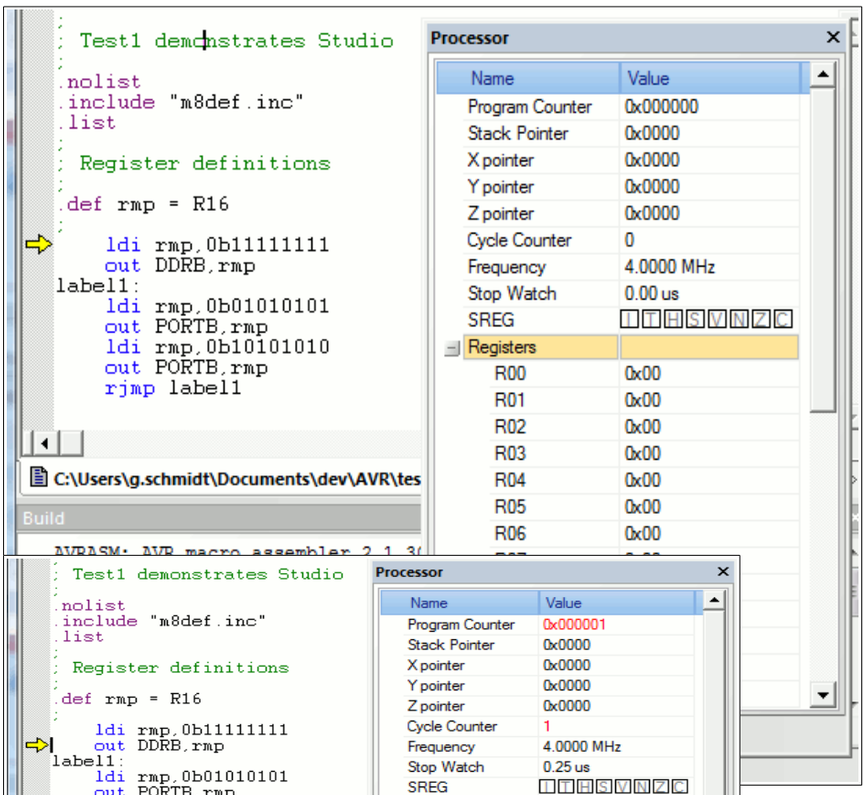
All window portions can be made larger and smaller and even can be shifted around on the screen. Try mixing these windows! The next pictures show some differently looking windows, but they are all the same as here.

After typing the source code shown above to your source file in the editor completely, push the menu “Build” and its sub-menu “Build”. If you typed correctly, the result shows up in your “Build” window.

Make sure, you read all window content once for the first time, because it gives you a lot more info besides the small green circle. All that should be fine, otherwise you typed errors into the code and the circle is red.

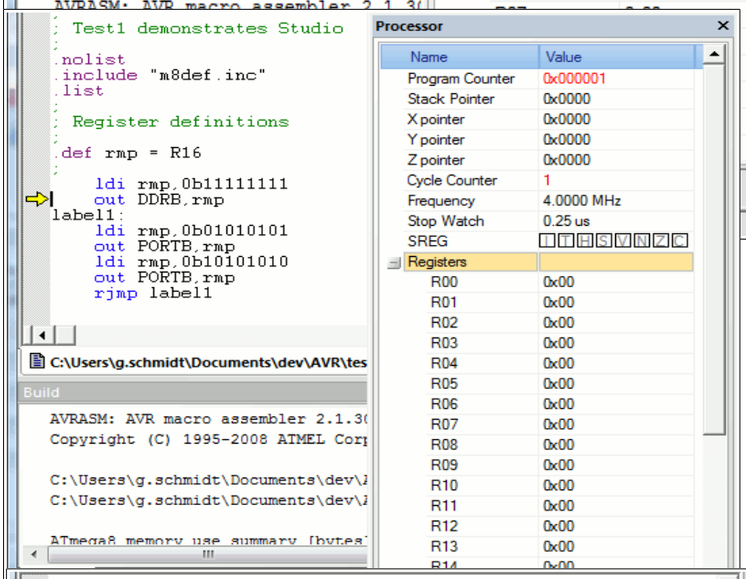
You can now push the menu item “Debug” and some windows change their content, size and position. If you also push the menu item “View”, “Toolbars” and “Processor” and shift around windows, it should look like this:

The former editor window has a yellow arrow now. This arrow points to the next instruction that will be executed (not really executed, but rather “simulated”).



The processor window shows the current program counter value (yes, the program starts at address 0), the stack pointer (no matter what that might be – wait for that later in the course), a cycle counter and a stop watch. If you push on the small "+" left to the word "Registers", the content of the 32 registers is displayed (yes, they are all empty when you start the processor simulation).

Now let us proceed with the first instruction. Menu item "Debug" and "Step into" or simply F11 executes the first instruction.



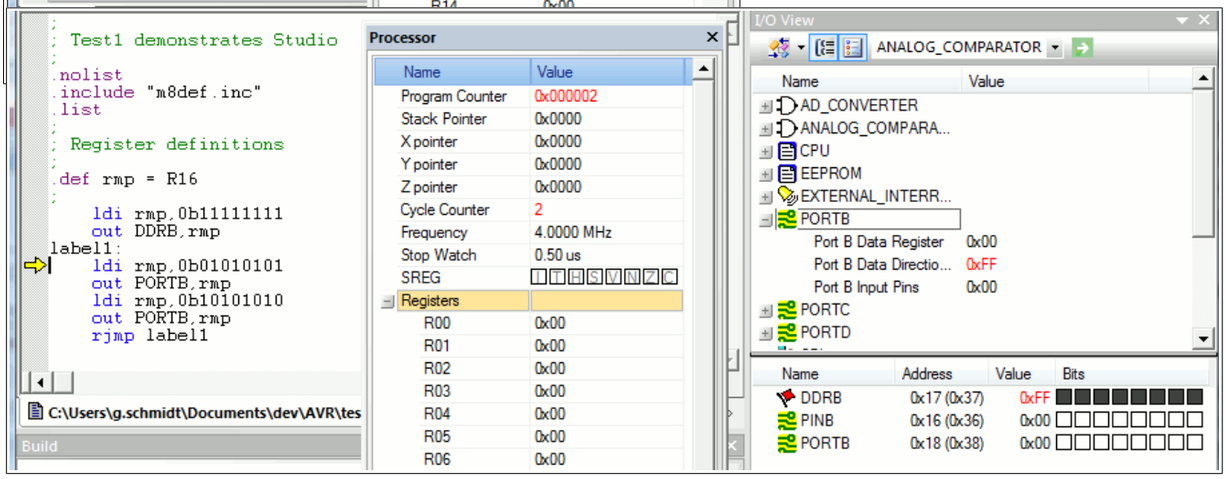
The instruction "ldi rmp,0b11111111" loads the binary value 1111.1111 to register R16. An instruction we will learn more about later on in the course.

The yellow arrow now has advanced one instruction down, is now at the OUT instruction.

In the processor window, the program counter and the cycle counter are both at 1 now.

And register 16, down the list of registers, is red now and shows 0xFF, which is hexadecimal for binary 1111.1111.

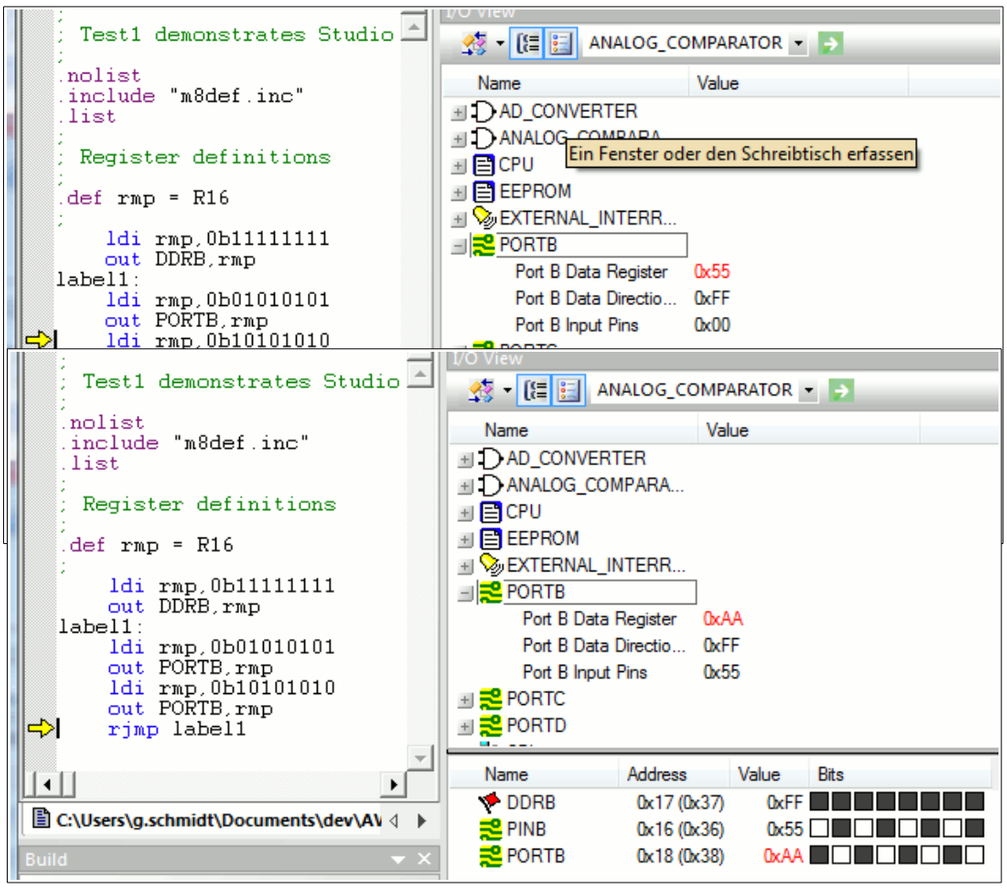
To learn about another simulator window just advance simulation one step further to execute the OUT instruction (e. g. by pushing the key F11.



The instruction "Out DDRB,rmp" writes 0xFF to a port named DDRB. Now the action is on the I/O view window. If you push on PORTB and the small "+" left of it, this window displays the value 0xFF in the port DDRB in two different forms: as 0xFF in the upper window portion and as 8 black squares in the lower window section.

To make it even more black, we push F11 two times and write 0x55 to the port PORTB.

As expected, the port PORTB changes its content and has four black and four white squares now.



Another two F11, writing 0xAA to PORTB, changes the black and white squares to the opposite color.

All what has been expected, but what happened to port PINB? We didn't write something to PINB, but it has the opposite colors than PORTB, just like the colors before in PORTB.

PINB is an input port for external pins. Because the direction ports in DDRB are set to be outputs, PINB follows the pin status of PORTB, just one cycle later. Nothing wrong here. If you like to check this, just press F11 several times and you see that this is correct.

That is our short trip through the simulator software world. The simulator is capable to much more, so it should be applied extensively in cases of design errors. Visit the different menu items, there is much more than can be shown here. In the mean time, instead of playing with the simulator, some basic things have to learned about assembler language, so put the Studio aside for a while.

5 What is a register?

Registers are special storages with 8 bits capacity and they look like this:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Note the numeration of these bits: the least significant bit starts with zero (mathematically: $2^0 = 1$).

A register can either store numbers from 0 to 255 (positive number, no negative values), or numbers from -128 to +127 (whole number with a sign bit, located in bit 7), or a value representing an ASCII-coded character (e. g. 'A'), or just eight single bits that do not have something to do with each other (e. g. for eight single flags, used to signal eight different yes/no decisions).

The special character of registers, compared to other storage sites, is that

- they are connected directly to the central processing unit called the accumulator,
- they can be used directly in assembler instructions, either as target register for the result or as read register for a calculation or transfer,
- operations with their content require only a single instruction word.

There are 32 registers in an AVR. They are originally named R0 to R31, but you can choose to name them to more meaningful ones using a so-called assembler directive. An example:

.DEF MyPreferredRegister = R16

Assembler directives always start with a dot. Instructions or labels do NEVER start with a dot. Note that assembler directives like this are only meaningful for the assembler but do not produce any code that is executable in the AVR target chip. The name “MyPreferredRegister” will not show up in the assembled hex code, and therefore this name cannot be derived from that hex code.

Instead of using the register name R16 we can now use our own name “MyPreferredRegister”, if we want to use R16 within an instruction. So we write a little bit more text each time we use this register, but we have an association what might be the content of this register.

Using the instruction line

```
LDI     MyPreferredRegister, 150
```

which means: load the number 150 immediately to the register R16, LoaD Immediate. This loads a fixed value or a constant to that register. Following the assembly, or translation of this code into binary or hex, the program storage written to the AVR chip looks like this:

000000 E906

This will show up in the listing, a file called *.lst produced by the assembler software, which is a simple text file. All numbers are in hex format: The first hex number is the address (000000), where the instruction is written to in the program flash memory of the AVR, the second is the instruction code (E906). E906 tells the processor three different things in one word, even if you don't see this directly:

- a basic load instruction code, that stands for LDI,
- the target register (R16) where the value 150 is to be written to,
- the value of the constant (150).

Don't be afraid: you don't have to remember this coding because the assembler knows how to translate all this to finally yield E906 and the AVR executes it.

Within one instruction two different registers can play a role. The easiest instruction of this type is the copy instruction, MOV. The naming of this instruction MOV deserves a price for the most confusing definition, because the content of a register cannot be moved (what would be left in a register, if you MOVE its content to somewhere else?). It should better be named COPY, because it copies the content of one register to another register. Like this:

```
.DEF MyPreferredRegister = R16
.DEF AnotherRegister = R15
        LDI MyPreferredRegister, 150
        MOV AnotherRegister, MyPreferredRegister
```

The first two lines of this monster program are directives that define the new names of the registers R16 and R15 for the assembler. Again, these lines do not produce any code for the AVR. The instruction lines with LDI and MOV produce code:

```
000000 E906
000001 2F01
```

The instruction write the value 150 into register R16 and copy its content to the target register R15. Very IMPORTANT NOTICE:

The first register is always the target register where the result is written to!

(This is unfortunately different from what one expects or from how we speak, think and write – left to right. It is a simple convention, probably inspired by some Asian languages where writing is from right to left. That was once defined that way to confuse the beginners learning assembler. That is why assembly language is that complicated.)

5.1 Different registers

The beginner might want to write the above instructions like this:

```
.DEF AnotherRegister = R15
        LDI AnotherRegister, 150
```

And: you lost. Only the registers from R16 to R31 load a constant immediately with the LDI instruction, R0 to R15 don't do that. This restriction is not very fine, but could not be avoided during construction of the instruction set for the AVRs.

There is one exception from that rule: setting a register to Zero. This instruction

```
        CLR MyPreferredRegister
```

is valid for all registers.

Besides the LDI instruction you will find this register class restriction with the following additional instructions:

- ANDI Rx,K ; Bit-And of register Rx with a constant value K,
- CBR Rx,M ; Clear all bits in register Rx that are set to one within the constant mask value M,
- CPI Rx,K ; Compare the content of the register Rx with a constant value K,
- SBCI Rx,K ; Subtract the constant K and the current value of the carry flag from the content of register Rx and store the result in register Rx,
- SBR Rx,M ; Set all bits in register Rx to one, that are one in the constant mask M,
- SER Rx ; Set all bits in register Rx to one (equal to LDI Rx,255),
- SUBI Rx,K ; Subtract the constant K from the content of register Rx and store the result in register Rx.

In all these instructions the register must be between R16 and R31! If you plan to use these instructions you should select one of these registers for that operation. It is shorter and easier to program. This is an additional reason why you should use the directive to define a register's name, because you can easier change the registers location later on, if required.

5.2 Pointer-registers

A very special extra role is defined for the register pairs R27:R26, R29:R28 and R31:R32. The role is so important that these pairs have extra short names in AVR assembler: X, Y and Z. These short names are understood by the assembler. These pairs are 16-bit pointer registers, able to point to addresses with max. 16 bit length, e. g. into SRAM locations (X, Y or Z) or into locations in program memory (Z).

5.2.1 Accessing memory locations with pointers

The lower byte of the 16-bit-address is located in the lower register, the higher byte in the upper register. Both parts have their own names, e. g. the higher byte of Z is named ZH (=R31), the lower Byte is ZL (=R30). These names are defined within the assembler. Dividing a 16-bit-word constant into its two different bytes and writing these bytes to a pointer register is done like follows:

```
.EQU address = RAMEND ; RAMEND is the highest 16-bit address in SRAM, defined in the *def.inc header file,
        LDI YH,HIGH(address) ; Load the MSB of address
        LDI YL,LOW(address) ; Load the LSB of address
```

Accesses via pointer registers are programmed with specially designed instructions. Read access is named LD (LoaD), write access named ST (STore), e. g. with the X-pointer:

Similarly you can use Y and Z for that purpose.

Pointer	***Sequence***	***Examples***
X	Read/Write from address X, don't change the pointer	LD R1,X or ST X,R1
X+	Read/Write from/to address X, and increment the pointer afterwards by one	LD R1,X+ or ST X+,R1
-X	First decrement the pointer by one and read/write from/to the new address afterwards	LD R1,-X or ST -X,R1

5.2.2 Reading program flash memory with the Z pointer

There is only one instruction for the read access to the program storage space. It is defined for the pointer pair Z and it is named LPM (Load from Program Memory). The instruction copies the byte at program flash address Z to the register R0. As the program memory is organized word-wise (one instruction on one address consists of 16 bits or two bytes or one word) the least significant bit selects the lower or upper byte (0=lower byte, 1= upper byte). Because of this the original address must be multiplied by 2 and access is limited to 15-bit or 32 kB program memory. Like this:

```
        LDI ZH,HIGH(2*address)
        LDI ZL,LOW(2*address)
        LPM
```

Following this instruction the address must be incremented to point to the next byte in program memory. As this is used very often a special pointer incrementation instruction has been defined to do this:

```
        ADIW ZL,1
        LPM
```

ADIW means ADd Immediate Word and a maximum of 63 can be added this way. Note that the assembler expects the lower of the pointer register pair ZL as first parameter. This is somewhat confusing as addition is done as 16-bit- operation.

The complement instruction, subtracting a constant value of between 0 and 63 from a 16-bit pointer register is named SBIW, Subtract Immediate Word. (SuBtract Immediate Word). ADIW and SBIW are possible for the pointer register pairs X, Y and Z and for the register pair R25:R24, that does not have an extra name and does not allow access to SRAM or program memory locations. R25:R24 is ideal for handling 16-bit values.

As incrementation after reading is very often needed, newer AVR types have the instruction

```
LPM R,Z+
```

This allows to transport the byte read to any location R, and auto-increments the pointer register.

5.2.3 Tables in the program flash memory

Now that you know how to read from flash memory you might wish to place a list of constants or a string of text to the flash and read these. How to insert that table of values in the program memory? This is done with the assembler directives .DB and .DW. With that you can insert byte wise or word wise lists of values. Byte wise organized lists look like this:

```
.DB 123,45,67,89 ; a list of four bytes, written in decimal form
.DB "This is a text. " ; a list of byte characters, written as text
```

You should always place an even number of bytes on each single line. Otherwise the assembler will add a zero byte at the end, which might be unwanted.

The similar list of words looks like this:

```
.DW 12345,6789 ; a list of two word constants
```

Instead of constants you can also place labels (e. g. jump targets) on that list, like that:

```
Label1:
[ ... here are some instructions ... ]
Label2:
[ ... here are some more instructions ... ]
Table:
.DW Label1,Label2 ; a word wise list of labels
```

Labels should start in column 1, but have to be ending with a ":". Note that reading the labels from that table with LPM (and subsequent incrementation of the pointer) first yields the lower byte of the word, then the upper byte.

5.2.4 Accessing registers with pointers

A very special application for the pointer registers is the access to the registers themselves. The registers are located in the first 32 bytes of the chip's address space (at address 0x0000 to 0x001F). This access is only meaningful if you have to copy the register's content to SRAM or EEPROM or read these values from there back into the registers. More common for the use of pointers is the access to tables with fixed values in the program memory space. Here is, as an example, a table with 10 different 16-bit values, where the fifth table value is read to R25:R24:

```
MyTable:
.DW 0x1234,0x2345,0x3456,0x4568,0x5678 ; The table values, word wise
.DW 0x6789,0x789A,0x89AB,0x9ABC,0xABCD ; organized
Read5: LDI ZH,HIGH(MyTable*2) ; address of table to pointer Z
        LDI ZL,LOW(MyTable*2) ; multiplied by 2 for bytewise access
        ADIW ZL,10 ; Point to fifth value in table
        LPM ; Read least significant byte from program memory
        MOV R24,R0 ; Copy LSB to 16-bit register
        ADIW ZL,1 ; Point to MSB in program memory
        LPM ; Read MSB of table value
        MOV R25,R0 ; Copy MSB to 16-bit register
```

This is only an example. You can calculate the table address in Z from some input value, leading to the respective table values. Tables can be organized byte- or character-wise, too.

5.3 Recommendation for the use of registers

The following recommendations, if followed, decide if you are an effective assembler programmer:

- Define names for registers with the .DEF directive, never use them with their direct name Rx.
- If you need pointer access reserve R26 to R31 for that purpose.
- A 16-bit-counter is best located in R25:R24.
- If you need to read from the program memory, e. g. fixed tables, reserve Z (R31:R30) and R0 for that purpose.
- If you plan to have access to single bits within certain registers (e. g. for testing flags), use R16 to R23 for that purpose.
- Registers necessary for math are best placed to R1 to R15.
- If you have more than enough registers available, place all your variables in registers.
- If you get short in registers, place as many variables as necessary to SRAM.

6 Ports

6.1 What is a Port?

Ports in the AVR are gates from the central processing unit to internal and external hard- and software components. The CPU communicates with these components, reads from them or writes to them, e. g. to the timers or the parallel ports. The most used port is the flag register, where flags from previous operations are written to and branching conditions are read from.

There are 64 different ports, which are not physically available in all different AVR types. Depending on the storage space and other internal hardware the different ports are either available and accessible or not. Which of the ports can be used in a certain AVR type is listed in the data sheets for the processor type. Larger ATmega and ATXmega have more than 64 ports, access to the ports beyond #63 is different then (see below).

Ports have a fixed address, over which the CPU communicates. The address is independent from the type of AVR. So e. g. the port address of port B is always 0x18 (0x stands for hexadecimal notation, 0x18 is decimal 24). You don't have to remember these port addresses, they have convenient aliases. These names are defined in the include files (header files) for the different AVR types, that are provided from the producer. The include files have a line defining port B's address as follows:

.EQU PORTB, 0x18

So we just have to remember the name of port B, not its location in the I/O space of the chip. The include file 8515def.inc is involved by the assembler directive

.INCLUDE "C:\Somewhere\8515def.inc"

and the registers of the 8515 are all defined there and easily accessible.

Ports usually are organized as 8-bit numbers, but can also hold up to 8 single bits that don't have much to do with each other. If these single bits have a meaning they have their own name associated in the include file, e. g. to enable the manipulation of a single bit. Due to that name convention you don't have to remember these bit positions. These names are defined in the data sheets and are given in the include file, too. They are provided here in the port tables.

6.2 Write access to ports

As an example the MCU General Control Register, called MCUCR, consists of a number of single control bits that control the general property of the chip. Here are the details of port MCUCR in the AT90S8515, taken from the device data book. Other ports look similar.

Bit	7	6	5	4	3	2	1	0	
$35 ($55)	**SRE**	**SRW**	**SE**	**SM**	**ISC11**	**ISC10**	**ISC01**	**ISC00**	**MCUCR**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

It is a port, fully packed with 8 control bits with their own names (ISC00, ISC01, ...). Those who want to send their AVR to a deep sleep need to know from the data sheet how to set the respective bits. Like this:

```
.DEF MyPreferredRegister = R16
        LDI MyPreferredRegister, 0b00100000
        OUT MCUCR, MyPreferredRegister
        SLEEP
```

The Out instruction brings the content of my preferred register, a Sleep-Enable-Bit called SE, to the port MCUCR. SE enables the AVR to go to sleep, whenever the SLEEP instruction shows up in the code. As all the other bits of MCUCR are also set by the above instructions and the Sleep Mode bit SM was set to zero, a mode called half-sleep will result: no further instruction execution will be performed but the chip still reacts to timer and other hardware interrupts. These external events interrupt the big sleep of the CPU if they feel they should notify the CPU.

The above formulation is not very transparent, because “0b00100000” is not easy to remember, and no one sees easily what bit exactly has been set to one by this instruction. So it is a good idea to formulate the LDI instruction as follows:

```
        LDI MyPreferredRegister, 1<<SE
```

This formulation tells the assembler to

- take a one (“1”),
- to read the bit position of the Sleep Enable bit (“SE”) from the symbol list, as defined in the header file 8515def.inc, which yields a value of “5” in that case,
- to shift (“<<”) the “1” five times left (“1<<5”), in steps:
    1. initial: 0000.0001,
    2. first shift left: 0000.0010,
    3. second shift left: 0000.0100, and so on until
    4. fifth shift left: 0010.0000.
- to associate this value to *MyPreferredRegister* and to insert this LDI instruction into the code.

To make it clear again: This shifting is done by the assembler software only, not within the code in the AVR. It is pure convention to increase the readability of the assembler source text.

How does this change, if you want to set the Sleep Mode bit (“SM”) and the Sleep Enable bit (“SE”) within the same LDI instruction? SM=1 and SE=1 enables your AVR to react to a SLEEP instruction by going to a big sleep, so only do this if you understand what the consequences are. The formulation is like this:

```
LDI MyPreferredRegister, (1<<SM) | (1<<SE)
```

Now, the assembler first calculates the value of the first bracket, (1<<SM), a “1” shifted four times left (because SM is 4) and that yields 0001.0000, then calculates the second bracket, (1<<SE), a “1” shifted five times left (because SE is 5). The “|” between the two brackets means BIT-OR the first and the second value, each bit one by one. The result of doing this with 0001.0000 and 0010.0000 in that case is 0011.0000, and that is our desired value for the LDI instruction. Even though the formulation

(1<<SM) | (1<<SE)

might, on the first look, not be more transparent than the resulting value

0011.0000

for a beginner, it is easier to understand which bits of MCUCR are intended to be manipulated in this LDI instruction. Especially if you have to read and understand your code some months later, SM and SE are a better hint that the Sleep Mode and Enable bits are targeted here. Otherwise you would have to consult the device's data book much more often.

6.3 Read access to ports

Reading a port's content is in most cases possible using the IN instruction. The following sequence

```
.DEF MyPreferredRegister = R16
        IN MyPreferredRegister, MCUCR
```

reads the bits in port MCUCR to the register named *MyPreferredRegister*. As many ports have undefined and unused bits in certain ports, these bits always read back as zeros.

More often than reading all 8 bits of a port one must react to a certain status bit within a port. In that case we don't need to read the whole port and isolate the relevant bit. Certain instructions provide an opportunity to execute instructions depending on the level of a certain bit of a port (see the JUMP section).

6.4 Read-Modify-Write access to ports

Setting or clearing certain bits of a port, without changing the other port bits, is also possible without reading and writing the other bits in the port. The two instructions are SBI (Set Bit I/O) and CBI (Clear Bit I/O). Execution is like this:

```
.EQU ActiveBit=0 ; The bit that is to be changed
        SBI PortB, ActiveBit ; The bit “ActiveBit” will be set to one
        CBI PortB, Activebit ; The bit “ActiveBit” will be cleared to zero
```

These two instructions have a limitation: only ports with an address smaller than 0x20 can be handled, ports above cannot be accessed that way. Because MCUCR in the above examples is at hex address $38, the sleep mode and enable bits can't be set or cleared that way. But all the port bits controlling external pins (PORTx, DDRx, PINx) are accessible that way.

6.5 Memory mapped port access

For the more exotic programmer and the “elephant-like” ATmega and ATXmega (where ATMEL ran out of accessible port addresses): the ports can also be accessed using SRAM access instructions, e. g. ST and LD. Just add 0x20 to the port's address (remember: the first 32 addresses are associated to the registers!) and access the port that way. Like demonstrated here:

```
.DEF MyPreferredRegister = R16
        LDI ZH,HIGH(PORTB+32)
        LDI ZL,LOW(PORTB+32)
        LD MyPreferredRegister,Z
```

That only makes sense in certain cases, because it requires more instructions, execution time and assembler lines, but it is possible. It is also the reason why the first address location of the SRAM is 0x60 or 0x100 in some larger AVR types.

6.6 Details of relevant ports in the AVR

The following table holds the most used ports in a “small” AT90S8515. Not all ports are listed here, some of the MEGA and AT90S4434/8535 types are skipped. If in doubt see the original reference.

Component	***Port name***	***Port-Register***
Accumulator	SREG	Status Register
Stack	SPL/SPH	Stackpointer
External SRAM/External Interrupt	MCUCR	MCU General Control Register
External Interrupts	GIMSK	Interrupt Mask Register
	GIFR	Interrupt Flag Register
Timer Interrupts	TIMSK	Timer Interrupt Mask Register
	TIFR	Timer Interrupt Flag Register
8-bit Timer 0	TCCR0	Timer/Counter 0 Control Register
	TCNT0	Timer/Counter 0

Component	*Port name*	*Port-Register*
16-bit Timer 1	TCCR1A	Timer/Counter Control Register 1 A
	TCCR1B	Timer/Counter Control Register 1 B
	TCNT1	Timer/Counter 1
	OCR1A	Output Compare Register 1 A
	OCR1B	Output Compare Register 1 B
	ICR1L/H	Input Capture Register
Watchdog Timer	WDTCR	Watchdog Timer Control Register
EEPROM Access	EEAR	EEPROM address Register
	EEDR	EEPROM Data Register
	EECR	EEPROM Control Register
Serial Peripheral Interface SPI	SPCR	Serial Peripheral Control Register
	SPSR	Serial Peripheral Status Register
	SPDR	Serial Peripheral Data Register
Serial Communication UART	UDR	UART Data Register
	USR	UART Status Register
	UCR	UART Control Register
	UBRR	UART Baud Rate Register
Analog Comparator	ACSR	Analog Comparator Control and Status Register
I/O-Ports	PORTx	Port Output Register
	DDRx	Port Direction Register
	PINx	Port Input Register

6.7 The status register as the most used port

By far the most often used port is the status register with its 8 bits. Usually access to this port is only by automatic setting and clearing bits by the CPU or accumulator, some access is by reading or branching on certain bits in that port, in a few cases it is possible to manipulate these bits directly (using the assembler instructions SEx or CLx, where x is the bit abbreviation). Most of these bits are set or cleared by the accumulator through bit-test, compare- or calculation-operations.

The most used bits are:

- Z: If set to one, the previous instruction yielded a zero result.
- C: If set to one, the previous instruction caused a carry of the most significant bit.

The following list has all assembler instructions that set or clear status bits depending on the result of the previous instruction execution.

Bit	*Calculation*	*Logic*	*Compare*	*Bits*	*Shift*	*Other*
Z	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR Z, BSET Z, CLZ, SEZ, TST	ASR, LSL, LSR, ROL, ROR	CLR
C	ADD, ADC, ADIW, SUB, SUBI, SBC, SBCI, SBIW	COM, NEG	CP, CPC, CPI	BCLR C, BSET C, CLC, SEC	ASR, LSL, LSR, ROL, ROR	-
N	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR N, BSET N, CLN, SEN, TST	ASR, LSL, LSR, ROL, ROR	CLR
V	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR V, BSET V, CLV, SEV, TST	ASR, LSL, LSR, ROL, ROR	CLR
S	SBIW	-	-	BCLR S, BSET S, CLS, SES	-	-
H	ADD, ADC, SUB, SUBI, SBC, SBCI	NEG	CP, CPC, CPI	BCLR H, BSET H, CLH, SEH	-	-
T	-	-	-	BCLR T, BSET T, BST, CLT, SET	-	-
I	-	-	-	BCLR I, BSET I, CLI, SEI	-	RETI

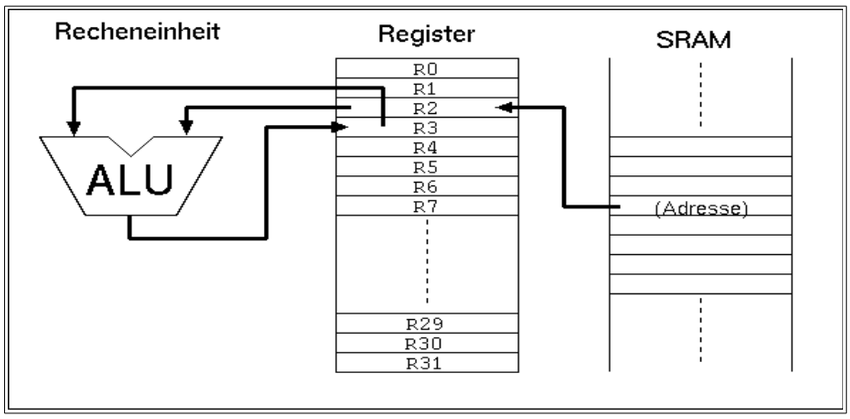
6.8 Port details

Port details of the most common ports are shown in an extra table (see annex).

7 SRAM

Nearly all AVR-types have static RAM (SRAM) on board (only very few old devices don't). Only very simple assembler programs can avoid using this memory space by putting all necessary information into registers. If you run out of registers you should be able to program the SRAM to utilize more space.

7.1 What is SRAM?



SRAM are memories that are not directly accessible by the central processing unit (Arithmetic and Logical Unit ALU, sometimes called accumulator) like the registers are. If you access these memory locations you usually use a register as interim storage. In the example displayed here a value in SRAM will be copied to the register R2 (1st instruction), a calculation with the value in R3 is made and the result is written to R3 (second instruction). After that this value is written back to the same SRAM location (instruction 3, not shown here).

So it is clear that operations with values stored in the SRAM are slower to perform than those using registers alone. On the other hand: even the smallest AVR types have 128 bytes of SRAM available, much more than the 32 registers can hold.

The types from the old AT90S8515 upwards offer the additional opportunity to connect additional external RAM, expanding the internal 512 bytes. From the assembler point-of-view, external SRAM is accessed like internal SRAM. No extra instructions must be learned for accessing that external SRAM.

7.2 For what purposes can I use SRAM?

Besides simple storage of values, SRAM offers additional opportunities for its use. Not only access with fixed addresses is possible, but also the use of pointers, so that floating access to subsequent locations in SRAM can be programmed. This way you can build up ring buffers for interim storage of values or calculated (variable) tables. This is not very often used with registers, because they are too few and prefer fixed access.

Even more relative is the access using an offset to a fixed starting address in one of the pointer registers. In that case a fixed address is stored in a pointer register, a constant value is added to this address and read/write access is made to that address with an offset. With that kind of access, tables are very more effective.

But the most relevant use for SRAM is the so-called stack. You can push values (variables) to that stack. Be it the content of a register, that is temporarily needed for another purpose. Be it a return address prior to calling a subroutine, or the return address prior to a hardware-triggered interrupt.

7.3 How to use SRAM?

7.3.1 Direct addressing

To copy a value to a memory location in SRAM you have to define the address. The SRAM addresses you can use reach from the start address (very often 0x0060 in smaller AVRs, 0x0100 in larger ATmega) to the end of the physical SRAM on the chip (in the AT90S8515 the highest accessible internal SRAM location is 0x025F, see the device data sheet of your AVR type for more details on this).

With the instruction

```
STS 0x0060, R1
```

the content of register R1 is copied to the first SRAM location in address 0x0060. With

```
LDS R1, 0x0060
```

the SRAM content at address 0x0060 is copied to the register. This is the direct access with an address that has to be defined by the programmer.

The symbols defined in the *def.inc include file, SRAM_START and RAMEND, allow to place your variables within the SRAM space. So it is better to use these definitions to access the 15th memory byte, like this:

```
LDS R1,SRAM_START+15
```

Symbolic names can be used to avoid handling fixed addresses, that require a lot of work, if you later want to change the structure of your data in the SRAM. These names are easier to handle than hex numbers, so give that address a name like:

```
.EQU MyPreferredStorageCell = SRAM_START
        STS MyPreferredStorageCell, R1
```

Yes, it isn't shorter, but easier to remember. Use whatever name that you find to be convenient.

7.3.2 Pointer addressing

Another kind of access to SRAM is the use of pointers. You need two registers for that purpose, that hold the 16-bit address of the location. As we learned in the Pointer-Register-Division, pointer registers are the register pairs X (XH:XL, R27:R26), Y (YH:YL, R29:R28) and Z (ZH:ZL, R31:R30). They allow access to the location they point to directly (e. g. with ST X, R1), after prior decrementing the address by one (e. g. ST -X, R1) or with subsequent auto-incrementation of the address (e. g. ST X+, R1). A complete access to three cells in a row looks like this:

```
.EQU MyPreferredStorageCell = SRAM_START
.DEF MyPreferredRegister = R1
.DEF AnotherRegister = R2
.DEF AndYetAnotherRegister = R3
        LDI XH, HIGH(MyPreferredStorageCell)
        LDI XL, LOW(MyPreferredStorageCell)
        LD MyPreferredRegister, X+
        LD AnotherRegister, X+
        LD AndYetAnotherRegister, X
```

Easy to operate, those pointers. And as easy as in other languages than assembler, that claim to be easier to learn.

7.3.3 Pointer with offset

The third construction is a little bit more exotic and only experienced programmers use this in certain cases. Let's assume we very often in our program need to access three consecutive SRAM locations. Let's further assume that we have a spare pointer register pair, so we can afford to use it exclusively for our purpose. If we would use the ST/LD instructions we always have to change the pointer if we access another location of the three. Not very convenient.

To avoid this, and to confuse the beginner, the access with offset was invented. During that access the register value isn't changed. The address is calculated by temporarily adding the fixed offset. In the above example the access to location 0x0062 would look like this. First, the pointer register is set to our central location SRAM_START:

```
.EQU MyPreferredStorageCell = SRAM_START
.DEF MyPreferredRegister = R1
        LDI YH, HIGH(MyPreferredStorageCell)
        LDI YL, LOW(MyPreferredStorageCell)
```

Somewhere later in the program I'd like to write to cell 2 above SRAM_START:

```
        STD Y+2, MyPreferredRegister
```

The corresponding instruction for reading from SRAM with an offset

```
        LDD MyPreferredRegister, Y+2
```

is also possible.

Note that the 2 is not really added to Y, just temporarily during the execution of this instruction. To confuse you further, this can only be done with the Y- and Z-register-pair, not with the X-pointer!

Of about 100 cases, the use of this opportunity is more effective in one single case. So don't care if you don't understand this in detail. It is only for experts, and only necessary in a few cases.

That's it with the SRAM, but wait: the most relevant use as stack is still to be learned.

7.4 Use of SRAM as stack

The most common use of SRAM is its use as stack. The stack is a tower of wooden blocks. Each additional block goes onto the top of the tower, each recall of a value removes the most upper block from the tower. Removal of blocks from the base or from any lower portion of the tower is too complicated and confuses your whole tower, so never try this. This structure is called Last-In-First-Out (LIFO) or easier: the last to go on top will be the first coming down from the top.

7.4.1 Defining SRAM as stack

To use SRAM as stack requires the setting of the stack pointer first. The stack pointer is a 16-bit-pointer, accessible like a port. The double register is named SPH:SPL. SPH holds the most significant address byte, SPL the least significant. This is only true, if the AVR type has more than 256 byte SRAM. If not, SPH is not necessary, is undefined, and must not and cannot be used. We assume we have more than 256 bytes SRAM in the following examples.

To construct the stack, the stack pointer is loaded with the highest available SRAM address. (In our case the tower grows downwards, towards lower addresses, just for historic reasons and to confuse the beginner!).

```
.DEF MyPreferredRegister = R16
        LDI MyPreferredRegister, HIGH(RAMEND) ; Upper byte
        OUT SPH,MyPreferredRegister ; to stack pointer
        LDI MyPreferredRegister, LOW(RAMEND) ; Lower byte
        OUT SPL,MyPreferredRegister ; to stack pointer
```

The value RAMEND is, of course, specific for the processor type. It is defined in the INCLUDE file for the processor type. The file 8515def.inc has the line:

```
.equ RAMEND =$25F ; Last On-Chip SRAM Location
```

The file 8515def.inc is included with the assembler directive

```
.INCLUDE "C:\somewhere\8515def.inc"
```

at the beginning of our assembler source code.

So we defined the stack now, and we don't have to care about the stack pointer any more, because manipulations of that pointer are mostly automatic.

7.4.2 Use of the stack

Using the stack is easy. The content of registers are pushed onto the stack like this:

```
PUSH MyPreferredRegister ; Throw that value on top of the stack
```

Where that value goes to is totally uninteresting. That the stack pointer was decremented after that push, we don't have to care. If we need the content again, we just add the following instruction:

```
POP MyPreferredRegister ; Read back the value from the top of the stack
```

With POP we just get the value that was last pushed on top of the stack. Pushing and popping registers makes sense, if

- the content is again needed some lines of the code later,
- all registers are in use, and if
- no other opportunity exists to store that value somewhere else.

If these conditions are not given, the use of the stack for saving registers is useless and just wastes processor time.

More sense makes the use of the stack in subroutines, where you have to return to the program location that called the routine. In that case the calling program code pushes the return address (the current program counter value) onto the stack and temporarily jumps to the subroutine. After its execution the subroutine pops the return address from the stack and loads it back into the program counter. Program execution is continued exactly one instruction behind the instruction, where the call happened:

```
        RCALL Somewhat ; Jump to the label "somewhat:"
[...] here we will later continue with the program.
```

Here the jump to the label "somewhat:" somewhere in the program code,

```
Somewhat: ; this is the jump address
[...] Here we do something
[...] and we are finished and want to jump back to the calling location:
        RET
```

During execution of the RCALL instruction the already incremented program counter, a 16-bit-address, is pushed onto the stack, using two pushes (the LSB and the MSB). By reaching the RET instruction, the content of the previous program counter is reloaded with two pops and execution continues there.

You don't need to care about the address of the stack, where the counter is loaded to. This address is automatically generated. Even if you call a subroutine within that subroutine the stack function is fine. This just packs two return addresses on top of the stack, the nested subroutine removes the first one, the calling subroutine the remaining one. As long as there is enough SRAM, everything is fine.

Servicing hardware interrupts isn't possible without the stack. Interrupts stop the normal execution of the program, wherever the program currently is. After execution of a specific service routine as a reaction to that interrupt program execution must return to the previous location, to before the interrupt occurred. This would not be possible if the stack is not able to store the return address.

The enormous advances of having a stack for interrupts are the reason, why even the smallest AVRs without having SRAM have at least a very small hardware stack.

7.4.3 Common bugs with the stack operation

For the beginner there are a lot of possible bugs, if you first learn to use stack.

Very clever is the use of the stack without first setting the stack pointer. Because this pointer is set to zero at program start, the pointer points to the location 0x0000, where register R0 is located. Pushing a byte results in a write to that register, overwriting its previous content. An additional push to the stack writes to 0xFFFF, an undefined position (if you don't have external SRAM there). A RCALL and RET will return to a strange address in program memory. Be sure: there is no warning, like a window popping up saying something like „Illegal access to memory location xxxx".

Another opportunity to construct bugs is to forget to pop a previously pushed value, or popping a value without pushing one first.

In a very few cases the stack overflows to below the first SRAM location. This happens in case of a never-ending recursive call. After reaching the lowest SRAM location the next pushes write to the ports (0x005F down to 0x0020), then to the registers (0x001F to 0x0000). Funny and unpredictable things happen with the chip hardware, if this goes on. Avoid this bug, it can even destroy your external hardware!

8 Jumping and branching

Here we discuss all instructions that control the sequential execution of a program. It starts with the starting sequence on power-up of the processor, continues with jumps, interrupts, etc.

8.1 Controlling sequential execution of the program

What happens during a reset?

When the power supply voltage of an AVR rises and the processor starts its work, the hardware triggers a reset sequence. The ports are set to their initial values, as defined in the device data sheet. The counter for the program steps will be set to zero. At this address the execution always starts. Here we have to have our first word of code. But not only during power-up this address is activated:

- During an external reset on the reset pin of the device a restart is executed.
- If the Watchdog counter reaches its maximum count, a reset is initiated. A watchdog timer is an internal clock that must be reseted from time to time by the program, otherwise it restarts the processor.
- You can call reset by a direct jump to that address (see the jump section below).

The third case is not a real reset, because the automatic resetting of register- and port-values to a well-defined default value is not executed. So, forget that for now.

The second option, the watchdog reset, must first be enabled by the program. It is disabled by default. Enabling requires write instructions to the watchdog's port. Setting the watchdog counter back to zero requires the execution of the instruction

```
WDR
```

to avoid a reset.

After execution of a reset, with setting registers and ports to default values, the code at address 0000 is word wise read to the execution part of the processor and is executed. During that execution the program counter is already incremented by one and the next word of code is already read to the code fetch buffer (Fetch during Execution). If the executed instruction does not require a jump to another location in the program the next instruction is executed immediately. That is why the AVRs execute extremely fast, each clock cycle executes one instruction (if no jumps occur).

The first instruction of an executable is always located at address 0000. To tell the compiler (assembler program) that our source code starts now and here, a special directive can be placed at the beginning, before the first code in the source is written:

```
.CSEG
.ORG 0000
```

The first directive, .CSEG, lets the compiler switch his output to the code section. All following is translated as code and is later written to the program flash memory section of the processor. Another target segment would be the EEPROM section of the chip, where you also can write bytes or words to.

```
.ESEG
```

The third segment is the SRAM section of the chip.

```
.DSEG
```

Other than with EEPROM content, where content is really going to the EEPROM during programming of the chip, the DSEG segment content is not programmed to the chip. There is no opportunity to burn any SRAM content. So the .DSEG is only used for correct label calculation during the assembly process. An example:

```
.DSEG ; The following are label definitions within the SRAM segment
MyFirstVariableIsAByte:
.BYTE 1 ; the DSEG-Pointer moves one byte upwards
MySecondVariableIsAWord:
.BYTE 2 ; the DSEG-Pointer moves two bytes upwards
MyThirdVariableIsAFieldForABuffer:
.BYTE 32; the DSEG-Pointer moves 32 bytes upwards
```

So, only three labels are defined within the assembler, no content is produced.

The ORG directive within the code segment, .ORG, above stands for the word “origin” and manipulates the address within the code segment, where assembled words go to. As our program always starts at 0x0000 the CSEG/ORG directives are trivial, you can skip these without getting into an error. We could start at 0x0100, but that makes no real sense as the processor starts execution at 0000. If you want to place a table exactly to a certain location of the code segment, you can use ORG. But be careful with that: Only jump forward with .ORG, never backwards. And be aware that the flash memory space that you skipped in between your current code location and the one you forced with .ORG is always filled with the instruction word 0xFFFF. This instruction does nothing, just goes to the next instruction. So be sure your execution never jumps into such undefined space in between.

If on the beginning of your code section you want to set a clear sign within your code, after first defining a lot of other things with .DEF- and .EQU-directives, use the CSEG/ORG sequence as a signal for yourself, even though it might not be necessary to do that.

As the first code word is always at address zero, this location is also called the reset vector. Following the reset vector the next positions in the program space, addresses 0x0001, 0x0002 etc., are interrupt vectors. These are the positions where the execution jumps to if an external or internal interrupt has been enabled and occurs. These positions called vectors are specific for each processor type and depend on the internal hardware available (see below). The instructions to react to such an interrupt have to be placed to the proper vector location. If you use interrupts, the first code, at the reset vector, must be a jump instruction, to jump over the other vectors. Each interrupt vector, that is planned to be enabled, must hold a jump instruction to the respective interrupt service routine. If the vector is not used, a dummy instruction like RETI

(RETurn from Interrupt) is best placed here. The typical program sequence at the beginning is like follows:

```
.CSEG
.ORG 0000
        RJMP Start ; the reset vector
        RJMP IntServRout1 ; the interrupt service routine for the first interrupt
        RETI ; a dummy for an unused interrupt
        RJMP IntServRout3 ; the interrupt service routine for the third interrupt
[...] here we place all the other interrupt vector instructions

[...] and here is a good place for the interrupt service routines themselves
IntServRout1:
   [...] Code of the first int service routine
        RETI ; end of service routine 1
IntServRout2:
   [...] Code of the third int service routine
        RETI ; end of service routine 2
[...] other code
Start: ; This here is the program start
[...] Here we place our main program
```

The instruction “RJMP Start” results in a jump to the label Start:, located some lines below. Remember, labels always end with a “:”. Labels, that don't fulfill these conditions are not taken for serious, but interpreted as instructions. Missing labels result in an error message ("Undefined label"), and compilation is interrupted.

8.2 Linear program execution and branches

Program execution is always linear, if nothing changes the sequential execution. These changes are the execution of an interrupt or of branching instructions.

Branching

Branching is very often depending on some condition, called conditional branching. As an example we assume we want to construct a 32-bit-counter using the registers R1 to R4. The least significant byte in R1 is incremented by one. If the register overflows during that operation (255 + 1 = 0), we have to increment R2 similarly. If R2 overflows, we have to increment R3, and so on.

Incrementation by one is done with the instruction INC. If an overflow occurs during that execution of INC R1, the zero bit in the status register is set to one (the result of the operation is zero). The carry bit in the status register, as usually set when something overflows, is not changed during an INC. This is not to confuse the beginner, but carry can be used for other purposes instead. The Zero-Bit or Zero-flag in this case is enough to detect an overflow. If no overflow occurs we can just leave the counting sequence.

If the Zero-bit is set, we must execute additional incrementation of the next upper register. To confuse the beginner the branching instruction, that we have to use, is not named BRNZ but BRNE (BRanch if Not Equal). A matter of taste ...

The whole count sequence of the 32-bit-counter should then look like this:

```
        INC R1 ; increase content of register R1
        BRNE GoOn32 ; if not zero, branch to GoOn32:
        INC R2 ; increase content of register R2
        BRNE GoOn32
        INC R3
        BRNE GoOn32
        INC R4
GoOn32:
```

So that's about it. An easy thing. The opposite condition to BRNE is BREQ or BRanch EQual.

Which of the status bits, also called processor flags, are changed during execution of an instruction is listed in instruction code tables, see the List of Instructions. Similarly to the Zero-bit you can use the other status bits like that:

```
BRCC  label/BRCS label; Carry-flag 0 (BRCC) or 1 (BRCS)
BRSH  label; Equal or greater
BRLO label; Smaller
BRMI label; Minus
BRPL label; Plus
BRGE label; Greater or equal (with sign bit)
BRLT label; Smaller (with sign bit)
BRHC label/BRHS label; Half overflow flag 0 or 1
BRTC label/BRTS label; T-Bit 0 or 1
BRVC label/BRVS label; Two's complement flag 0 or 1
BRIE label/BRID label; Interrupt enabled or disabled
```

to react to the different conditions. Branching always occurs if the condition is met. Don't be afraid, most of these instructions are rarely used. For the beginner only Zero and Carry are relevant.

8.3 Timing during program execution

Like mentioned above the required time to execute one instruction is equal to the processor's clock cycle. If the processor runs on a 4 MHz clock frequency then one instruction requires 1/4 µs or 250 ns, at 10 MHz clock only 100 ns. The required time is as exact as the internal or external or Xtal clock is. If you need exact timing an AVR is the optimal solution for your problem. Note that there are a few instructions that require two or more cycles, e. g. the branching instructions (if branching occurs) or the SRAM read/write sequence. See the instruction table for details.

To define exact timing there must be an opportunity that does nothing else than delay program execution. You might use

other instructions that do nothing, but more clever is the use of the no-operation instruction NOP. This is the most useless instruction:

```
        NOP
```

This instruction does nothing but wasting processor time. At 4 MHz clock we need just four of these instructions to waste 1 µs. No other hidden meanings here on the NOP instruction. For a signal generator with 1 kHz we don't need to add 4000 such instructions to our source code, but we use a software counter and some branching instructions. With these we construct a loop that executes for a certain number of times and are exactly delayed. A counter could be a 8-bit-register that is decremented with the DEC instruction, e. g. like this:

```
        CLR R1 ; one clock cycle
Count:
        DEC R1 ; one clock cycle
        BRNE Count ; two for branching, one for not branching
```

This sequence wastes (1) + (255*3) + (1*2) = 768 clock cycles or 192 µs at 4 MHz.

16-bit counting can also be used to delay exactly, like this

```
        LDI ZH,HIGH(65535) ; one clock cycle
        LDI ZL,LOW(65535) ; one clock cycle
Count:
        SBIW ZL,1 ; two clock cycles
        BRNE Count ; two for branching, one for not branching
```

This sequence wastes (1+1) + (65534*4) + (1*3) = 262,141 clock cycles or 65,535.25 µs at 4 MHz.

If you use more registers to construct nested counters you can reach any delay. And the delay is as exact as your clock source is, even without a hardware timer.

8.4 Macros and program execution

Very often you have to write identical or similar code sequences on different occasions in your source code. If you don't want to write it once and jump to it via a subroutine call you can use a macro to avoid getting tired writing the same sequence several times. Macros are code sequences, designed and tested once, and inserted into the code by its macro name. As an example we assume we need to delay program execution several times by 1 µs at 4 MHz clock. Then we define a macro somewhere in the source:

```
.MACRO Delay1
        NOP
        NOP
        NOP
        NOP
.ENDMACRO
```

This definition of the macro does not yet produce any code, it is silent. Code is produced only if you call that macro by its name:

```
[...] somewhere in the source code
        Delay1
[...] code goes on here
```

This results in four NOP instructions inserted to the code at that location. An additional “Delay1” inserts additional four NOP instructions.

If your macro has longer code sequences, or if you are short in code storage space, you should avoid the use of macros and use subroutines instead.

By calling a macro by its name you can add some parameters to manipulate the produced code. But this is more than a beginner has to know about macros.

8.5 Subroutines

In contrary to macros a subroutine does save program storage space. The respective sequence is only once stored in the code and is called from whatever part of the code. To ensure continued execution of the sequence following the subroutine call you need to return to the caller. For a delay of 10 cycles you need to write this subroutine:

```
Delay10: ; the call of the subroutine requires some cycles
        NOP ; delay one cycle
        NOP ; delay one cycle
        NOP ; delay one cycle
        RET ; return to the caller
```

Subroutines always start with a label, otherwise you would not be able to jump to it, here named “Delay10:”. Three NOPs follow and a RET instruction. If you count the necessary cycles you just find 7 cycles (3 for the NOPs, 4 for the RET). The missing 3 are for calling that routine:

```
[...] somewhere in the source code:
        RCALL Delay10
[...] further on with the source code
```

RCALL is a relative call. The call is coded as relative jump, the relative distance from the calling routine to the subroutine is calculated by the compiler. The RET instruction jumps back to the calling routine. Note that before you use subroutine calls you must set the stack pointer (see Stack), because the return address must be packed on top of the stack during the RCALL instruction.

If you want to jump directly to somewhere else in the code you have to use the jump instruction:

```
[...] somewhere in the source code
        RJMP Delay10
Return:
[...] further on with source code
```

Note that RJMP is also a relative jump instruction with limited distance. Only ATmega AVRs have a JMP instruction allowing jumps over the complete flash memory space, but these instructions require two words and more instruction time than RJMP, so avoid it if possible.

The routine that you jumped to can not use the RET instruction in that case, because RJMP does not place the current execution address to the stack. To return back to the calling location in the source requires to add another label and the called routine to jump back to this label. Jumping like this is not like calling a subroutine because you can't call this routine from different locations in the code.

RCALL and RJMP are unconditioned branches. To jump to another location, depending on some condition, you have to combine these with branching instructions. Conditioned calling of a subroutine can best be done with the following (confusing) instructions. If you want to call a subroutine depending on a certain bit in a register use the following sequence:

```
SBRC R1,7 ; Skip the next instruction if bit 7 in register 1 is 0
RCALL UpLabel ; Call that subroutine
```

SBRC reads „Skip next instruction if Bit 7 in Register R1 is Clear (=Zero)“. The RCALL instruction to “UpLabel:” is only executed if bit 7 in register R1 is 1, because the next instruction is skipped if it would be 0. If you like to call the subroutine in case this bit is 0 then you use the corresponding instruction SBRS. The instruction following SBRS/SBRC can be a single word or double word instruction, the processor knows how far he has to jump over it. Note that execution times are different then. To jump over more than one following instruction these instructions cannot be used.

If you have to skip an instruction in case two registers have the same value you can use the following exotic instruction:

```
CPSE R1,R2 ; Compare R1 and R2, skip next instruction if equal
RCALL SomeSubroutine ; Call SomeSubroutine
```

A rarely used instruction, forget it for the beginning. If you like to skip the following instruction depending on a certain bit in a port use the following instructions SBIC and SBIS. That reads “Skip if the Bit in I/o space is Clear (or Set)”, like this:

```
SBIC PINB,0 ; Skip next instruction if Bit 0 on input port B is 0
RJMP ATarget ; Jump to the label ATarget
```

The RJMP-instruction is only executed if bit 0 in port B is high. This is something confusing for the beginner. The access to the port bits is limited to the lower half of ports, the upper 32 ports are not usable here.

Now, another exotic application for the expert. Skip this if you are a beginner. Assume we have a bit switch with 4 switches connected to port B. Depending on the state of these 4 bits we would like to jump to 16 different locations in the code. Now we can read the port and use several branching instructions to find out, where we have to jump to today. As alternative you can write a table holding the 16 addresses, like this:

```
MyTab:
        RJMP Routine1
        RJMP Routine2
        [...]
        RJMP Routine16
```

In our code we copy that address of the table to the Z pointer register:

```
LDI ZH,HIGH(MyTab)
LDI ZL,LOW(MyTab)
```

and add the current state of the port B (in R16) to this address.

```
        ADD ZL,R16
        BRCC NoOverflow
        INC ZH
NoOverflow:
```

Now we can jump to this location in the table, either for calling a subroutine:

```
ICALL ; call the subroutine which address is in Z
```

or as a jump with no way back:

```
IJMP ; jump to address in Z
```

The processor loads the content of the Z register pair into its program counter and continues operation there. More clever than branching over and over?

8.6 Interrupts and program execution

Very often we have to react on hardware conditions or other events. An example is a change on an input pin. You can program such a reaction by writing a loop, asking whether a change on the pin has occurred. This method is called polling, its like a bee running around in circles searching for new flowers. If there are no other things to do and reaction time does not matter, you can do this with the processor. If you have to detect short pulses of less than a µs duration this method is useless. In that case you need to program an interrupt.

An interrupt is triggered by some hardware conditions. All hardware interrupts are disabled at reset time by default, so the condition has to be enabled first. The respective port bits enabling the component's interrupt ability are set first. The processor has a bit in its status register enabling him to respond to the interrupt of all components, the Interrupt Enable Flag. Enabling the general response to interrupts requires the following instruction:

```
SEI ; Set Int Enable Bit
```

Each single interrupt requires additional port manipulation to be enabled.

If the interrupting condition occurs, e. g. a change on the port bit, the processor pushes the actual program counter to the stack (which must be enabled first! See initiation of the stackpointer in the Stack section of the SRAM description). Without that, the processor wouldn't be able to return back to the location, where the interrupt occurred (which could be any time and anywhere within program execution). After that, processing jumps to the predefined location, the interrupt vector, and executes the instructions there. Usually the instruction there is a JUMP instruction to the interrupt service routine, located somewhere in the code. The interrupt vector is a processor-specific location and depending from the hardware component and the condition that leads to the interrupt. The more hardware components and the more conditions, the more vectors. The different vectors for some older AVR types are listed in the following table. (The first vector isn't an interrupt but the reset vector, performing no stack operation!)

Name	***Interrupt Vector Address***			***Triggered by***
	2313	***2323***	***8515***	
RESET	0000	0000	0000	Hardware Reset, Power-On-Reset, Watchdog Reset
INT0	0001	0001	0001	Level change on the external INT0 pin
INT1	0002	-	0002	Level change on the external INT1 pin
TIMER1CAPT	0003	-	0003	Capture event on Timer/Counter 1
TIMER1COMPA	-	-	0004	Timer/Counter 1 = Compare value A
TIMER1 COMPB	-	-	0005	Timer/Counter 1 = Compare value B
TIMER1 COMP1	0004	-	-	Timer/Counter 1 = Compare value 1
TIMER1 OVF	0005	-	0006	Timer/Counter 1 Overflow
TIMER0 OVF	0006	0002	0007	Timer/Counter 0 Overflow
SPI STC	-	-	0008	Serial Transmit Complete
UART TX	0007	-	0009	UART char in receive buffer available
UART UDRE	0008	-	000A	UART transmitter ran empty
UART TX	0009	-	000B	UART All Sent
ANA_COMP	-	-	000C	Analog Comparator

Note that the capability to react to events is very different for the different types. The addresses are sequential, but not identical for different types. Consult the data sheet for each AVR type.

The higher a vector in the list the higher is its priority. If two or more components have an interrupt condition pending at the same time, the up most vector with the lower vector address wins. The lower int has to wait until the upper int was served. To disable lower ints from interrupting during the execution of its service routine the first executed int disables the processor's I-flag. The service routine must re-enable this flag after it is done with its job.

For re-setting the I status bit there are two ways. The service routine can end with the instruction:

```
RETI
```

This return from the int routine restores the I-bit after the return address has been loaded to the program counter.

The second way is to enable the I-bit by the instruction

```
SEI ; Set Interrupt Enabled
RET ; Return
```

This is not the same as the RETI, because subsequent interrupts are already enabled before the program counter is re-loaded with the return address. If another int is pending, its execution is already starting before the return address is popped from the stack. Two or more nested addresses remain on the stack. No bug is to be expected, but it is an unnecessary risk doing that. So just use the RETI instruction to avoid this unnecessary flow to the stack.

An Int-vector can only hold a relative jump instruction to the service routine. If a certain interrupt is not used or undefined we can just put a RETI instruction there, in case an erroneously enabled int happens before we wrote an interrupt service routine. In a few cases it is absolutely necessary to react to these false ints. That is the case if the execution of the respective service routine does not automatically reset the interrupt condition flag of the peripheral. In that case a simple RETI would reset the otherwise never-ending interrupts. This is the case with some of the UART interrupts.

Note that larger devices have a two-word organization of the vector table. In this case the JMP instruction has to be used instead of RJMP. And RETI instructions must be followed by an NOP to point to the next vector table address.

As, after an interrupt is under service, further execution of lower-priority interrupts are blocked, all int service routines should be as short as possible. If you need to have a longer routine to serve the int, use one of the two following methods. The first is to allow ints by SEI within the service routine, whenever you're done with the most urgent tasks. This is not very clever. More convenient is to perform the urgent tasks, setting a flag somewhere in a register for the slower reaction portions and return from the int immediately.

A very serious rule for int service routines is:

The first instruction is always to save the processor status flags in a register or on the stack.

Do this before you use instructions that might change flags in the status flag register. The reason is that the interrupted main program might just be in a state using the flag for a branch decision, and the int would just change that flag to another state. Funny things would happen from time to time. The last instruction before the RETI therefore is to copy the saved flags from the register back to status port or to pop the status register content from the stack and restore its original content. The following shows examples how to do that:

Saving in a register:

```
Isr:
```

Saving on the stack:

```
Isr:
```

```
IN R15,SREG ; save flags
[... more instructions...]

OUT SREG,R15 ; restore flags

RETI ; return from interrupt
```

```
PUSH R15 ; save register on stack
IN R15, SREG
[...more instructions...]
OUT SREG,R15 ; restore flags
POP R15
RETI ; return from interrupt
```

The method on the right is slower, the method on the left requires a register exclusively for that purpose.

Generally: All used registers in a service routine should either be exclusively reserved for that purpose or saved on stack and restored at the end of the service routine. Never change the content of a register within an int service routine that is used somewhere else in the normal program without restoring it.

Because of these basic requirements a more sophisticated example for an interrupt service routine here.

```
.CSEG ; Code-Segment starts here
.ORG 0000 ; Address is zero
        RJMP Start ; The reset-vector on Address 0000
        RJMP IService ; 0001: first Int-Vector, INT0 service routine
[...] here other vectors
Start: ; Here the main program starts
[...] here is enough space for defining the stack and other things

IService: ; Here we start with the Interrupt-Service-Routine
        PUSH R16 ; save a register to stack
        IN R16,SREG ; read status register
        PUSH R16 ; and put on stack
[...] Here the Int-Service-Routine does something and uses R16
        POP R16 ; get previous flag register from stack
        OUT SREG,R16 ; restore old status
        POP R16 ; get previous content of R16 from the stack
        RETI ; and return from int
```

Looks a little bit complicated, but is a prerequisite for using ints without producing serious bugs. Skip PUSH R16 and POP R16 if you can afford reserving the register for exclusive use within the service routine. As an interrupt service routine cannot be interrupted (unless you allow interrupts within the routine), all different int service routines can use the same register.

You understand now, why allowing interrupts within an interrupt service routine, and not at its end with RETI, is not a good idea?

That's it for the beginner. There are some other things with ints, but this is enough to start with, and not to confuse you.

9 Calculations

Here we discuss all necessary instructions for calculating in AVR assembler language. This includes number systems, setting and clearing bits, shift and rotate, and adding/subtracting/comparing and the format conversion of numbers.

9.1 Number systems in assembler

The following formats of numbers are common in assembler:

- Positive whole numbers (Bytes, Words, Longwords, etc.),
- Signed whole numbers (ShortInts, Integers, LongInts, etc.),
- Binary Coded Digits (BCD),
- Packed BCDs,
- ASCII-formatted numbers.

If you come from a high-level language: forget pre-defined number formats. Assembler doesn't have that concept nor its (sometimes frustating) limitations. What you earn is: you are the master of your own format!

9.1.1 Positive whole numbers (bytes, words, etc.)

The smallest whole number to be handled in assembler is a byte with eight bits. This codes numbers between 0 and 255. Such bytes fit exactly into one register of the MCU. All larger numbers must be based on this basic format, using more than one register. Two bytes yield a word (range from 0 .. 65,535), three bytes form a longer word (range from 0 .. 16,777,215) and four bytes form a double word (range from 0 .. 4,294,967,295).

The single bytes of a word or a double word can be stored in whatever register you prefer. Operations with these single bytes are programmed byte by byte, so you don't have to put them in a row. In order to form a row for a double word we could store it like this:

```
.DEF r16 = dw0
.DEF r17 = dw1
.DEF r18 = dw2
.DEF r19 = dw3
```

Registers dw0 to dw3 are in a row, but don't need to be. If we need to initiate this double word at the beginning of an application (e. g. to 4,000,000), this should look like this:

```
.EQU dwi = 4000000 ; define the constant
        LDI dw0,LOW(dwi) ; The lowest 8 bits to R16
        LDI dw1,BYTE2(dwi) ; bits 8 .. 15 to R17
        LDI dw2,BYTE3(dwi) ; bits 16 .. 23 to R18
        LDI dw3,BYTE4(dwi) ; bits 24 .. 31 to R19
```

So we have splitted this decimal number, called dwi, to its binary portions BYTE4 to BYTE1 and packed them into the four byte packages. Now you can calculate with this double word.

9.1.2 Signed numbers (integers)

Sometimes, but in rare cases, you need negative numbers to calculate with. A negative number is defined by interpreting the most significant bit of a byte as sign bit. If it is 0 the number is positive. If it is 1 the number is negative. If the number is negative we usually do not store the rest of the number as is, but we use its inverted value. Inverted means that -1 as a byte integer is not written as 1000.0001 but as 1111.1111 instead. That means: subtract 1 from 0 (and forget the overflow). The first bit is the sign bit, signaling that this is a negative number. Why this different format (subtracting the number from 0) is used is easy to understand: adding -1 (1111.1111) and +1 (0000.0001) yields exactly zero, if you forget the overflow that occurs during that operation (to the ninth bit).

In one byte the largest integer number to be handled is +127 (binary 01111111), the smallest one is -128 (binary 1,0000000). In other computer languages this number format is called short integer. If you need a bigger range of values you can add another byte to form a larger integer value, ranging from +32,767 .. -32,768), four bytes provide a range from +2,147,483,647 .. -2,147,483,648, in other languages called a LongInt or DoubleInt.

9.1.3 Binary Coded Digits, BCD

Positive or signed whole numbers in the formats discussed above use the available space most effectively. Another, less dense number format, but easier to handle and understand is to store decimal numbers in a byte for one digit each. The decimal digit is stored in its binary form in a byte. Each digit from 0 .. 9 needs four bits (binary values 0000 .. 1001), the upper four bits of the byte are always zeros, blowing a lot of hot air into one byte. For to handle the value 250 we would need at least three bytes, e. g.:

Bit value	*128*	*64*	*32*	*16*	*8*	*4*	*2*	*1*
R16, Digit 1 =2	0	0	0	0	0	0	1	0
R17, Digit 2 = 5	0	0	0	0	0	1	0	1
R18, Digit 3 = 0	0	0	0	0	0	0	0	0

```
;Instructions to use:
LDI R16,2
LDI R17,5
LDI R18,0
```

You can calculate with these numbers, but this is a bit more complicated in assembler than calculating with binary values. The advantage of this format is that you can handle as long numbers as you like, as long as you have enough storage space. The calculations are as precise as you like (if you program AVRs for banking applications), and you can convert them very easily to character strings.

9.1.4 Packed BCDs

If you pack two decimal digits into one byte you don't loose that much storage space. This method is called packed binary coded digits. The two parts of a byte are called upper and lower nibble. The upper nibble usually holds the more significant digit, which has advantages in calculations (special instructions in AVR assembler language). The decimal number 250 would look like this when formatted as a packed BCD:

Byte	*Digits*	*Value*	*8*	*4*	*2*	*1*	*8*	*4*	*2*	*1*
2	4 & 3	02	0	0	0	0	0	0	1	0
1	2 & 1	50	0	1	0	1	0	0	0	0

```
; Instructions for setting:
LDI R17,0x02 ; Upper byte
LDI R16,0x50 ; Lower byte
```

To set this correct you can use the binary notation (0b...) or the hexadecimal notation (0x...) to set the proper bits to their correct nibble position.

Calculating with packed BCDs is a little more complicated compared to the binary form. Format changes to character strings are nearly as easy as with BCDs. Length of numbers and precision of calculations is only limited by the storage space.

9.1.5 Numbers in ASCII-format

Very similar to the unpacked BCD format is to store numbers in ASCII format. The digits 0 to 9 are stored using their ASCII (ASCII = American Standard Code for Information Interchange) representation. ASCII is a very old format, developed and optimized for teletype writers, unnecessarily very complicated for computer use (do you know what a char named End Of Transmission EOT meant when it was invented?), very limited in range for other than US languages (only 7 bits per character), still used in communications today due to the limited efforts of some operating system programmers to switch to more effective character systems. This ancient system is only topped by the European 5-bit long teletype character set called Baudot set or the Morse code, still used by some finger-nervous people.

Within the ASCII code system the decimal digit 0 is represented by the number 48 (hex 0x30, binary 0b0011.0000), digit 9 is 57 decimal (hex 0x39, binary 0b0011.1001). ASCII wasn't designed to have these numbers on the beginning of the code set as there are already instruction chars like the above mentioned EOT for the teletype. So we still have to add 48 to a BCD (or set bit 4 and 5 to 1) to convert a BCD to ASCII. ASCII formatted numbers need the same storage space like BCDs. Loading 250 to a register set representing that number would look like this:

```
LDI R18,'2'
LDI R17,'5'
LDI R16,'0'
```

The ASCII representation of these characters are written to the registers.

9.2 Bit manipulations

To convert a BCD coded digit to its ASCII representation we need to set bit 4 and 5 to a one. In other words we need to OR the BCD with a constant value of hex 0x30. In assembler this is done like this:

```
ORI R16,0x30
```

If we have a register that is already set to hex 0x30 we can use the OR with this register to convert the BCD:

```
OR R1,R2
```

Back from an ASCII character to a BCD is as easy. The instruction

```
ANDI R16,0x0F
```

isolates the lower four bits (= the lower nibble). Note that ORI and ANDI are only possible with registers above R15. If you need to do this, use one of the registers R16 to R31!

If the hex value 0x0F is already in register R2, you can AND the ASCII character with this register:

```
AND R1,R2
```

The other instructions for manipulating bits in a register are also limited for registers above R15. They would be formulated

like this:

```
SBR R16,0b00110000 ; Set bits 4 and 5 to one
CBR R16,0b00110000 ; Clear bits 4 and 5 to zero
```

If one or more bits of a byte have to be inverted you can use the following instruction (which is not possible for use with a constant):

```
LDI R16,0b10101010 ; Invert all uneven bits
EOR R1,R16 ; in register R1 and store result in R1
```

To invert all bits of a byte is called the One's complement:

```
COM R1
```

inverts the content in register R1 and replaces zeros by one and vice versa. Different from that is the Two's complement, which converts a positive signed number to its negative complement (subtracting from zero). This is done with the instruction

```
NEG R1
```

So +1 (decimal: 1) yields -1 (binary 1.1111111), +2 yields -2 (binary 1.1111110), and so on.

Besides the manipulation of the bits in a register, copying a single bit is possible using the so-called T-bit of the status register. With

```
BST R1,0
```

the T-bit is loaded with a copy of bit 0 in register R1. The T-bit can be set or cleared, and its content can be copied to any bit in any register:

```
CLT ; clear T-bit, or
SET ; set T-bit, or
BLD R2,2 ; copy T-bit to register R2, bit 2
```

9.3 Shift and rotate

Shifting and rotating of binary numbers means multiplying and dividing them by 2. Shifting has several sub-instructions.

Multiplication with 2 is easily done by shifting all bits of a byte one binary digit left and writing a zero to the least significant bit. This is called logical shift left or LSL. The former bit 7 of the byte will be shifted out to the carry bit in the status register.

```
LSL R1
```

The inverse division by 2 is the instruction called logical shift right, LSR.

```
LSR R1
```

The former bit 7, now shifted to bit 6, is filled with a 0, while the former bit 0 is shifted into the carry bit of the status register. This carry bit could be used to round up and down (if set, add one to the result). Example, division by four with rounding:

```
    LSR R1 ; division by 2
    BRCC Div2 ; Jump if no round up
    INC R1 ; round up
Div2:
    LSR R1 ; Once again division by 2
    BRCC DivE ; Jump if no round up
    INC R1 ; Round Up
DivE:
```

So, dividing is easy with binaries as long as you divide by multiples of 2.

If signed integers are used the logical shift right would overwrite the sign-bit in bit 7. The instruction „arithmetic shift right“ ASR leaves bit 7 untouched and shifts the 7 lower bits, inserting a zero into bit location 6.

```
ASR R1
```

Like with logical shifting the former bit 0 goes to the carry bit in the status register.

What about multiplying a 16-bit word by 2? The most significant bit of the lower byte has to be shifted to yield the lowest bit of the upper byte. In that step a shift would set the lowest bit to zero, but we need to shift the carry bit from the previous shift of the lower byte into bit 0 of the upper byte. This is called a rotate. During rotation the carry bit in the status register is shifted to bit 0, the former bit 7 is shifted to the carry during rotation.

```
LSL R1 ; Logical Shift Left of the lower byte
ROL R2 ; ROtate Left of the upper byte
```

The logical shift left in the first instruction shifts bit 7 to carry, the ROL instruction rolls it to bit 0 of the upper byte. Following the second instruction the carry bit has the former bit 7 of the upper byte. The carry bit can be used to either indicate an overflow (if 16-bit-calculation is performed) or to roll it into more upper bytes (if more than 16 bit calculation is done).

Rolling to the right is also possible, dividing by 2 and shifting carry to bit 7 of the result:

```
LSR R2 ; Logical Shift Right, bit 0 to carry
ROR R1 ; ROtate Right and shift carry in bit 7
```

It's easy dividing with big numbers. You see that learning assembler is not THAT complicated.

The last instruction that shifts four bits in one step is very often used with packed BCDs. This instruction shifts a whole nibble from the upper to the lower position and vice versa. In our example we need to shift the upper nibble to the lower nibble position. Instead of using

```
ROR R1
ROR R1
ROR R1
ROR R1
```

we can perform that with a single

```
SWAP R1
```

This instruction exchanges the upper and lower nibble. Note that the content of the upper nibble will be different after applying these two methods.

9.4 Adding, subtracting and comparing

The following calculation operations are too complicated for the beginners and demonstrate that assembler is only for extreme experts, hi. Read on your own risk!

9.4.1 Adding and subtracting 16-bit numbers

To start complicated we add two 16-bit-numbers in R1:R2 and R3:R4. (In this notation, we mean that the first register is the most significant byte, the second the least significant).

```
ADD R2,R4 ; first add the two low-bytes
ADC R1,R3 ; then the two high-bytes
```

Instead of a second ADD we use ADC in the second instruction. That means add with carry, which is set or cleared during the first instruction, depending from the result. Already scared enough by that complicated math? If not: take this!

We subtract R3:R4 from R1:R2.

```
SUB R2,R4 ; first the low-byte
SBC R1,R3 ; then the high-byte
```

Again the same trick: during the second instruction we subtract another 1 from the result if the result of the first instruction had an overflow. Still breathing? If yes, handle the following!

9.4.2 Comparing 16-bit numbers

Now we compare a 16-bit-word in R1:R2 with the one in R3:R4 to evaluate whether it is bigger than the second one. Instead of SUB we use the compare instruction CP, instead of SBC we use CPC:

```
CP R2,R4 ; compare lower bytes
CPC R1,R3 ; compare upper bytes
```

If the carry flag is set now, R1:R2 is smaller than R3:R4.

9.4.3 Comparing with constants

Now we add some more complicated stuff. We compare the content of R16 with a constant: 0b10101010.

```
CPI R16,0xAA
```

If the Zero-bit in the status register is set after that, we know that R16 is equal to 0xAA. If the carry-bit is set, we know, it is smaller. If Carry is not set and the Zero-bit is not set either, we know it is larger.

And now the most complicated test. We evaluate whether R1 is zero or negative:

```
TST R1
```

If the Z-bit is set, the register R1 is zero and we can follow with the instructions BREQ, BRNE, BRMI, BRPL, BRLO, BRSH, BRGE, BRLT, BRVC or BRVS to branch around a little bit.

9.4.4 Packed BCD math

Still with us? If yes, here is some packed BCD calculations. Adding two packed BCDs can result in two different overflows. The usual carry shows an overflow, if the higher of the two nibbles overflows to more than 15 decimal. Another overflow, from the lower to the upper nibble occurs, if the two lower nibbles add to more than 15 decimal.

To take an example we add the packed BCDs 49 (=hex 49) and 99 (=hex 99) to yield 148 (=hex 0x0148). Adding these in binary math, results in a byte holding hex 0xE2, no byte overflow occurs. The lower of the two nibbles should have an overflow, because 9+9=18 (more than 9) and the lower nibble can only handle numbers up to 15. The overflow was added to bit 4, the lowest significant bit of the upper nibble. Which is correct! But the lower nibble should be 8 and is only 2 (18 = 0b0001.0010). We should add 6 to that nibble to yield a correct result. Which is quite logic, because whenever the lower nibble reaches more than 9 we have to add 6 to correct that nibble.

The upper nibble is totally incorrect, because it is 0xE and should be 3 (with a 1 overflowing to the next upper digit of the packed BCD). If we add 6 to this 0xE we get to 0x4 and the carry is set (=0x14). So the trick is to first add these two numbers and then add 0x66 to correct the 2 digits of the packed BCD. But halt: what if adding the first and the second number would not result in an overflow to the next nibble? And not result in a digit above 9 in the lower nibble? Adding 0x66 would then result in a totally incorrect result. The lower 6 should only be added if the lower nibble either overflows to the upper nibble or results in a digit larger than 9. The same with the upper nibble.

How do we know, if an overflow from the lower to the upper nibble has occurred? The MCU sets the H-bit in the status

register, the half-carry bit. The following shows the algorithm for the different cases that are possible after adding two nibbles and adding hex 0x6 after that.

1. Add the nibbles. If overflow occurs (C for the upper nibbles, or H for the lower nibbles), add 6 to correct, if not, do step 2.
2. Add 6 to the nibble. If overflow occurs (C resp. H), you're done. If not, subtract 6.

To program an example we assume that the two packed BCDs are in R2 and R3, R1 will hold the overflow, and R16 and R17 are available for calculations. R16 is the adding register for adding 0x66 (the register R2 cannot add a constant value), R17 is used to correct the result depending from the different flags. Adding R2 and R3 goes like that:

```
        LDI R16,0x66 ; for adding 0x66 to the result
        LDI R17,0x66 ; for later subtracting from the result
        ADD R2,R3 ; add the two two-digit-BCDs
        BRCC NoCy1 ; jump if no byte overflow occurs
        INC R1 ; increment the next higher byte
        ANDI R17,0x0F ; don't subtract 6 from the higher nibble
NoCy1:
        BRHC NoHc1 ; jump if no half-carry occurred
        ANDI R17,0xF0 ; don't subtract 6 from lower nibble
NoHc1:
        ADD R2,R16 ; add 0x66 to result
        BRCC NoCy2 ; jump if no carry occurred
        INC R1 ; increment the next higher byte
        ANDI R17,0x0F ; don't subtract 6 from higher nibble
NoCy2:
        BRHC NoHc2 ; jump if no half-carry occurred
        ANDI R17,0xF0 ; don't subtract 6 from lower nibble
NoHc2:
        SUB R2,R17 ; subtract correction
```

A little bit shorter than that:

```
        LDI R16,0x66
        ADD R2,R16
        ADD R2,R3
        BRCC NoCy
        INC R1
        ANDI R16,0x0F
NoCy:
        BRHC NoHc
        ANDI R16,0xF0
NoHc:
        SUB R2,R16
```

Question to think about: Why is that equally correct, half as long and less complicated and where is the trick?

9.5 Format conversion for numbers

All number formats can be converted to any other format. The conversion from BCD to ASCII and vice versa was already shown above (Bit manipulations).

9.5.1 Conversion of packed BCDs to BCDs, ASCII or Binaries

Conversion of packed BCDs is not very complicated either. First we have to copy the number to another register. With the copied value we change nibbles using the SWAP instruction to exchange the upper and the lower one. The upper part is cleared, e. g. by ANDing with 0x0F. Now we have the BCD of the upper nibble and we can either use as is (BCD) or set bit 4 and 5 to convert it to an ASCII character. After that we copy the byte again and treat the lower nibble without first SWAPping and get the lower BCD.

A little bit more complicated is the conversion of BCD digits to a binary. Depending on the numbers to be handled we first clear the necessary bytes that will hold the result of the conversion. We then start with the highest BCD digit. Before adding this to the result we multiply the result with 10. (Note that in the first step this is not necessary, because the result is zero either).

In order to do the multiplication by 10, we copy the result to somewhere else. Then we multiply the result by four (two left shifts resp. rolls). Adding the previously copied number to this yields a multiplication with 5. Now a multiplication with 2 (left shift/roll) yields the 10-fold of the result. Finally we add the BCD and repeat that algorithm until all decimal digits are converted. If, during one of these operations, there occurs a carry of the result, the BCD is too large to be converted. This algorithm handles numbers of any length, as long as the result registers are prepared.

9.5.2 Conversion of Binaries to BCD

The conversion of a binary to BCDs is more complicated than that. If we convert a 16-bit-binary we can subtract 10,000 (0x2710), until an overflow occurs, yielding the first digit. Then we repeat that with 1,000 (0x03E8) to yield the second digit. And so on with 100 (0x0064) and 10 (0x000A), then the remainder is the last digit. The constants 10,000, 1,000, 100 and 10 can be placed to the program memory storage in a word wise organized table, like this:

```
DezTab:
.DW 10000, 1000, 100, 10
```

and can be read word-wise with the LPM instruction from the table.

An alternative is a table that holds the decimal value of each bit in the 16-bit-binary, e. g.

```
.DB 0,3,2,7,6,8
.DB 0,1,6,3,8,4
.DB 0,0,8,1,9,2
.DB 0,0,4,0,9,6
.DB 0,0,2,0,4,8 ; and so on until
.DB 0,0,0,0,0,1
```

Then you shift the single bits of the binary left out of the registers to the carry. If it is a one, you add the number in the table to the result by reading the numbers from the table using LPM. This is more complicated to program and a little bit slower than the above method.

A third method is to calculate the table value, starting with 000001, by adding this BCD with itself, each time after you have shifted a bit from the binary to the right, and added to the BCD result.

Many methods, much to optimize here.

9.6 Multiplication

Multiplication of binary numbers is explained here.

9.6.1 Decimal multiplication

In order to multiply two 8-bit-binaries we remind ourselves, how this is done with decimal numbers:

```
    1234 * 567 = ?
------------------------
    1234 *   7 =    8638
+   1234 *  60 =   74040
+   1234 * 500 =  617000
------------------------
    1234 * 567 =  699678
========================
```

In single steps decimal:

- We multiply the first number with the lowest significant digit of the second number and add this to the result.
- We multiply the first number with 10 and then with the next higher digit of the second number and add this to the result.
- We multiply the first number with 100, then with the third-highest digit, and add this to the result.

9.6.2 Binary multiplication

Now in binary. Multiplication with the single digits is not necessary, because there are only the digits 1 (add the number) and 0 (don't add the number). Multiplication by 10 in decimal goes to multiplication by 2 in binary mode. Multiplication by 2 is done easily, either by adding the number with itself, or by shifting all bits one position left and writing a 0 to the void position on the right. You see that binary math is very much easier than decimal. Why didn't mankind use this from the beginning?

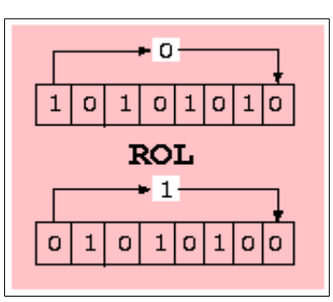
9.6.3 AVR assembler program

The following source code demonstrates realization of multiplication in assembler.

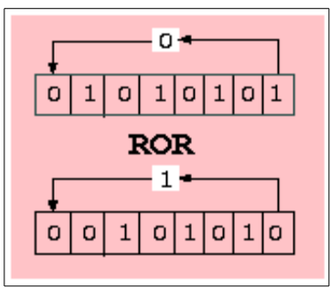
```
; Mult8.asm multiplies two 8-bit-numbers to yield a 16-bit-result
;
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc"
.LIST
;
; Flow of multiplication
;
; 1.The binary to be multiplicated with is shifted bitwise into the carry bit. If it is a one, the binary number is added to the
;   result, if it is not a one that was shifted out, the number is not added.
; 2.The binary number is multiplied by 2 by rotating it one position left, shifting a 0 into the void position.
; 3.If the binary to be multiplied with is not zero, the multiplication loop is repeated. If it is zero, the multiplication is done.
;
; Used registers
;
.DEF rm1 = R0 ; Binary number to be multiplicated (8 Bit)
.DEF rmh = R1 ; Interim storage
.DEF rm2 = R2 ; Binary number to be multiplicated with (8 Bit)
.DEF rel = R3 ; Result, LSB (16 Bit)
.DEF reh = R4 ; Result, MSB
.DEF rmp = R16 ; Multi purpose register for loading
;
.CSEG
.ORG 0000
;
```

```
        rjmp START
;
START:
        ldi rmp,0xAA ; example binary 1010.1010
        mov rm1,rmp  ; to the first binary register
        ldi rmp,0x55 ; example binary 0101.0101
        mov rm2,rmp  ; to the second binary register
;
; Here we start with the multiplication of the two binaries in rm1 and rm2, the result will go to reh:rel (16 Bit)
;
MULT8:
;
; Clear start values
        clr rmh ; clear interim storage
        clr rel ; clear result registers
        clr reh
;
; Here we start with the multiplication loop
;
MULT8a:
;
; Step 1: Rotate lowest bit of binary number 2 to the carry flag (divide by 2, rotate a zero into bit 7)
;
        clc ; clear carry bit
        ror rm2 ; bit 0 to carry, bit 1 to 7 one position to the right, carry bit to bit 7
;
; Step 2: Branch depending if a 0 or 1 has been rotated to the carry bit
;
        brcc MULT8b ; jump over adding, if carry has a 0
;
; Step 3: Add 16 bits in rmh:rml to the result, with overflow from LSB to MSB
;
        add rel,rm1 ; add LSB of rm1 to the result
        adc reh,rmh ; add carry and MSB of rm1
;
MULT8b:
;
; Step 4: Multiply rmh:rm1 by 2 (16 bits, shift left)
;
        clc ; clear carry bit
        rol rm1 ; rotate LSB left (multiply by 2)
        rol rmh ; rotate carry into MSB and MSB one left
;
; Step 5: Check if there are still one's in binary 2, if yes, go on multiplicating
;
        tst rm2 ; all bits zero?
        brne MULT8a ; if not, go on in the loop
;
; End of the multiplication, result in reh:rel
;
; Endless loop
;
LOOP:
        rjmp loop
```

9.6.4 Binary rotation

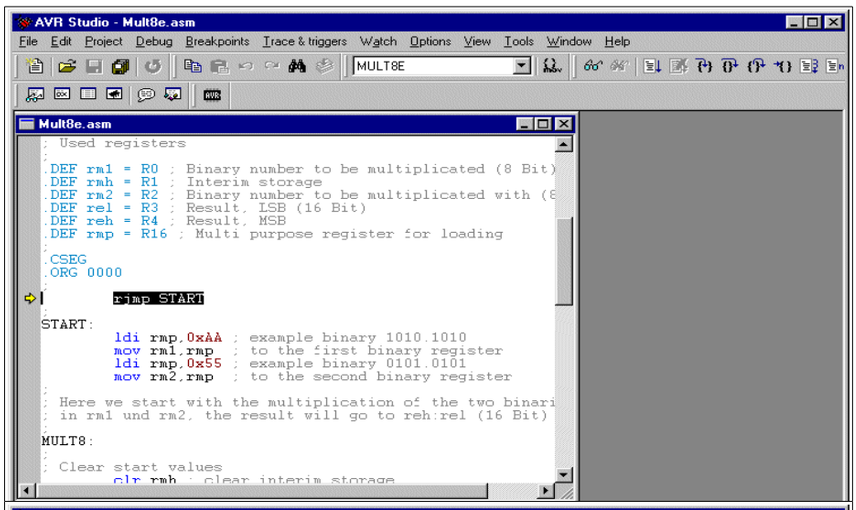


For understanding the multiplication operation, it is necessary to understand the binary rotation instructions ROL and ROR. These instructions shift all bits of a register one position left (ROL) resp. right (ROR). The void position in the register is filled with the content of the carry bit in the status register, the bit that rolls out of the register is shifted to this carry bit. This operation is demonstrated using 0xAA as an example for ROL and 0x55 as an example for ROR.

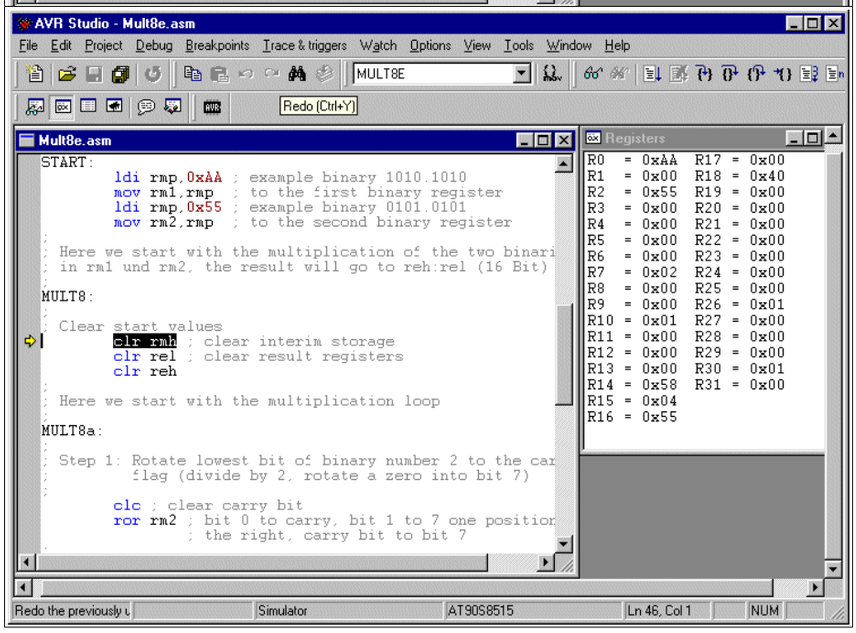


9.6.5 Multiplication in the studio

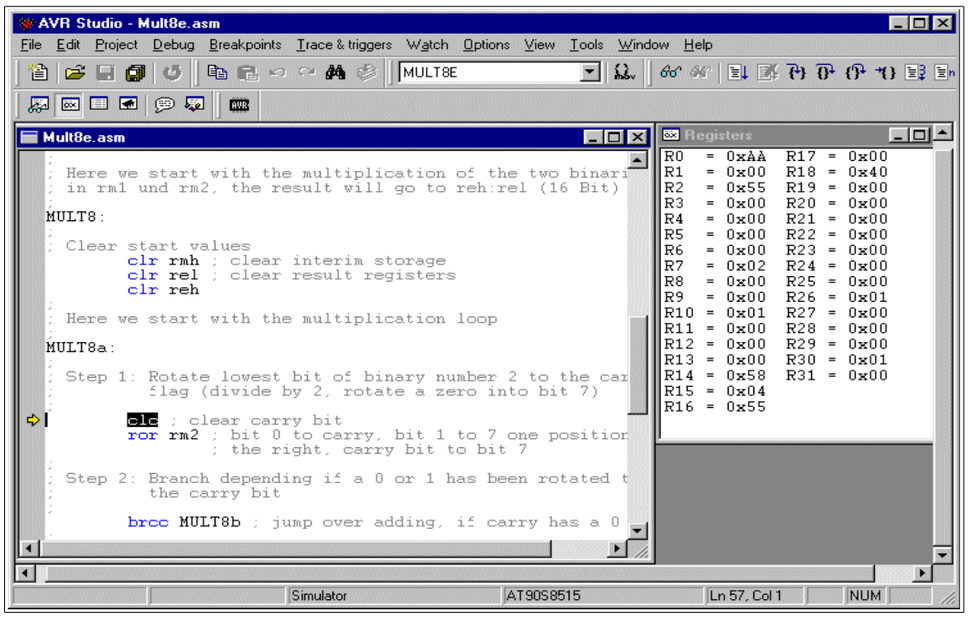
The following screen shots show the multiplication program in the simulator (to make a difference: here Studio version 3).



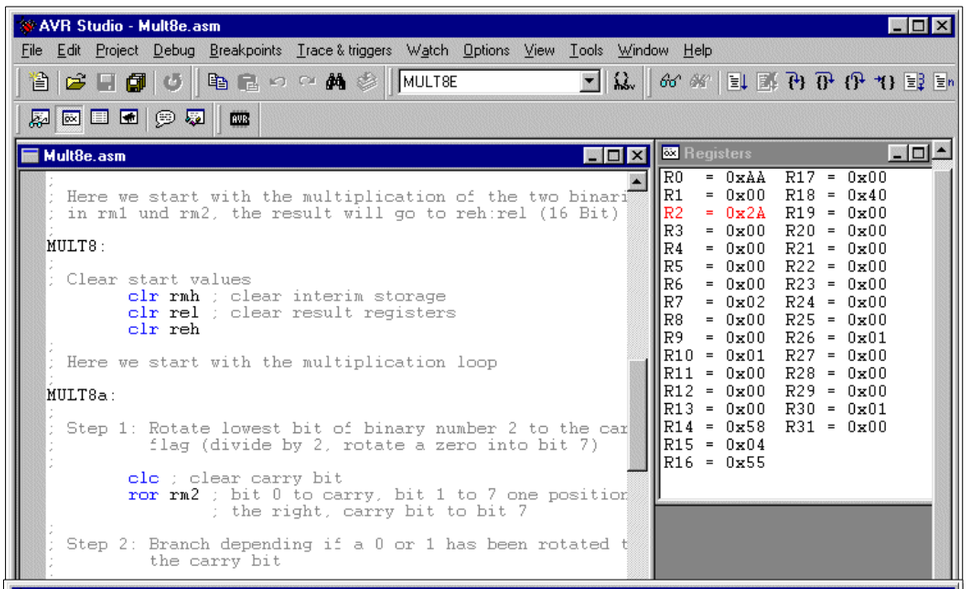
The object-code has been opened, the cursor is placed on the first executable instruction. F11 does single steps.



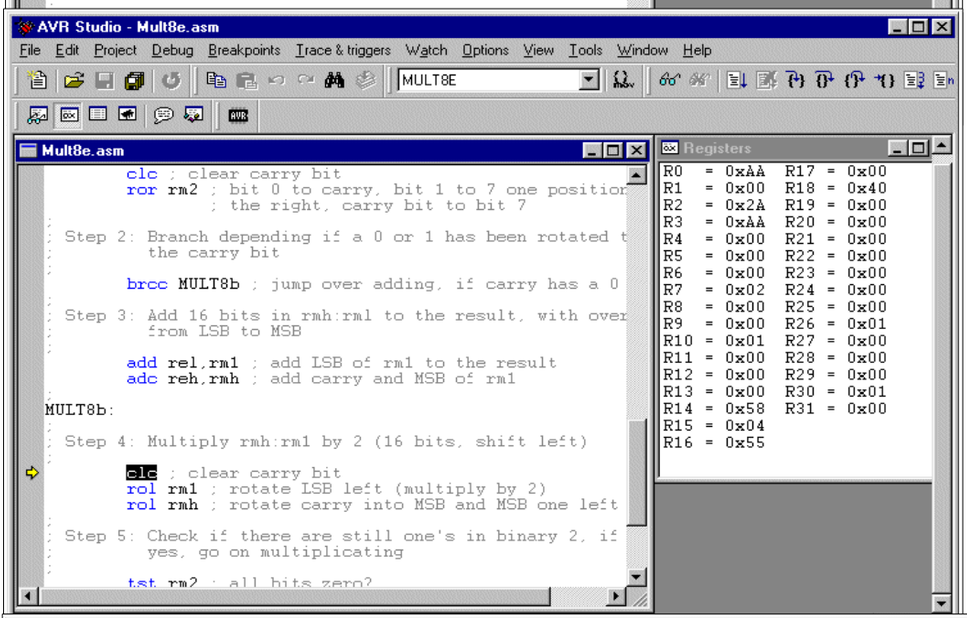
The registers R0 and R2 are set to 0xAA and 0x55, our test binaries, to be multiplied.



R2 is rotated to the right, to roll the least significant bit into the carry bit. 0x55 (0101.0101) yielded 0x2A (0010.1010).

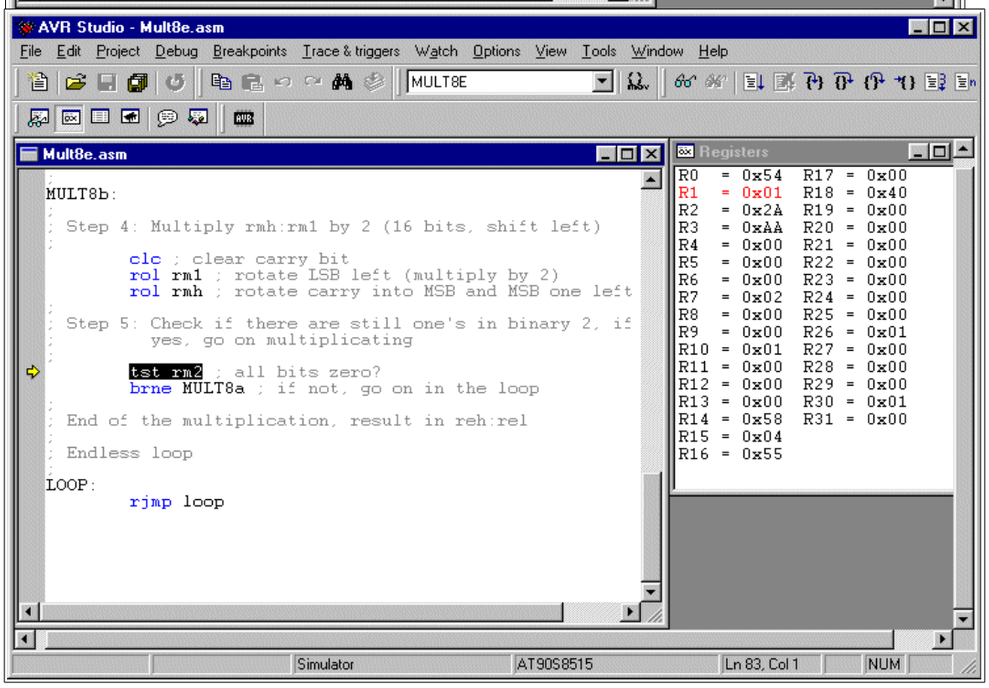


Because the carry bit had a one, the content of the registers R1:R0 is added to the (empty) register pair R4:R3, resulting in 0x00AA there.



Now the register pair R1:R0 is rotated one position left to multiply this binary by 2. From 0x00AA, multiplication by 2 yields 0x0154.

The whole multiplication loop is repeated as long there is at least one binary 1 in register R2. These following loops are not shown here.



Using key F5 of the studio we multi-stepped over these loops to a breakpoint at the end of the multiplication routine. The result register pair R4:R3 has the result of the multiplication of 0xAA by 0x55: 0x3872.

This wasn't that complicated, just remind yourself on the similar decimal operations. Binary multiplication is much easier than decimal.

9.7 Hardware multiplication

All ATmega, ATXmega, AT90CAN and AT90PWM have an on-board hardware multiplicator, that performs 8 by 8 bit multiplications in only two clock cycles. So whenever you have to do multiplications and you are sure that this software never ever needs not to run on an AT90S- or ATtiny-chip, you can make use of this hardware feature.

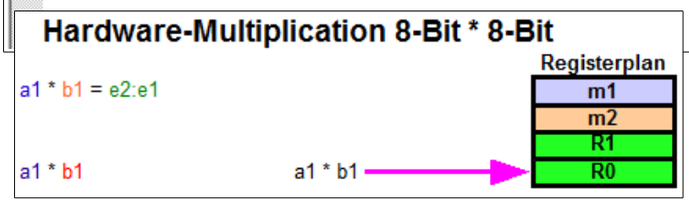
The following shows how to multiply

- 8-by-8-binaries,
- 16-by-8-binaries,
- 16-by-16-binaries,
- 16-by-24-binaries.

9.7.1 Hardware multiplication of 8-by-8-bit binaries

The use is simple and straight-forward: if the two binaries to be multiplied are in the registers R16 and R17, just type

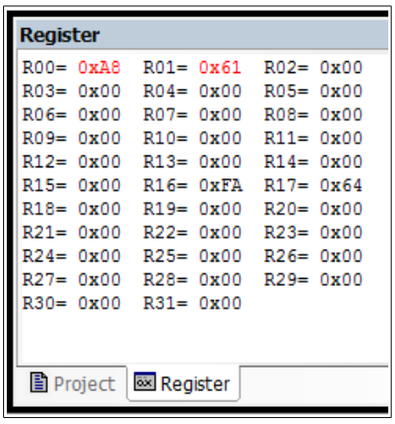
```
; Tests 8-by-8-bit hardware multiplication with ATmega8
;
; Define Registers
;
.def ResL = R0
.def ResM = R1
.def m1 = R16
.def m2 = R17
;
; Load multiplicators
;
    ldi m1,250
    ldi m2,100
;
; Perform multiplication
;
    mul m1,m2
;
; 16-bit result is in R1:R0
;
```

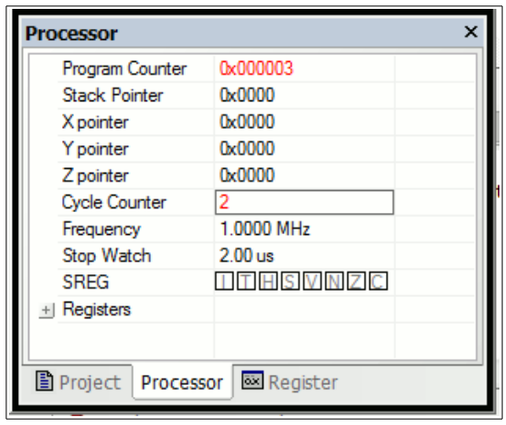


mul R16,R17

As the result of these two 8-bit binaries might be up two 16 bits long, the result will be in the registers R1 (most significant byte) and R0 (least significant byte). That's all about it.

The program demonstrates the simulation in the Studio. It multiplies decimal 250 (hex FA) by decimal 100 (hex 64), in the registers R16 and R17.





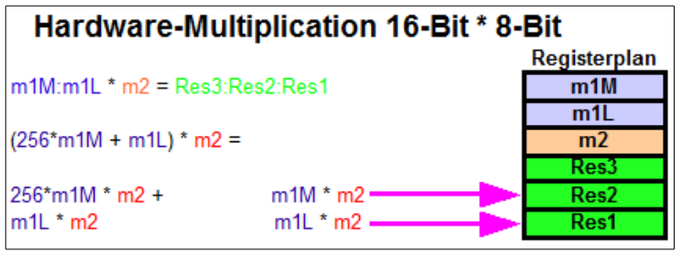
After execution, the registers R0 (LSB) and R1 (MSB) hold the result hex 61A8 or decimal 25,000.

And: yes, that requires only two cycles, or 2 microseconds with a 1 Mcs/s clock.

9.7.2 Hardware multiplication of a 16- by an 8-bit-binary

You have a larger binary to multiply? Hardware is limited to 8, so we need to invest some genius ideas instead. To solve the problem with larger binaries, we just look at this combination of 16 and 8 first. Understanding this concept helps understanding the method, so you will be able to solve the 32-by-64-bit multiplication problem later.

First the math: a 16-bit-binary m1M:m1L are simply two 8-bit-binaries m1M and m1L, where the most significant one m1M of these two is multiplied by decimal 256 or hex 100. (For those who need a reminder: the decimal 1234 is simply (12 multiplied by 100) plus 34, or (1 multiplied by 1000) plus (2 multiplied by 100) plus (3 multiplied by 10) plus 4.



So the 16-bit-binary m1 is equal to 256*m1M plus m1L, where m1M is the MSB and m1L is the LSB. Multiplying m1 by 8-bit-binary m2 so is, mathematically formulated:

- m1 * m2 = (256*m1M + m1L) * m2, or
- 256*m1M*m2 + m1L*m2.

So we just need to do two multiplications and to add both results. Sorry, if you see three asterisks in the formula: the multiplication with 256 in the binary world doesn't require any hardware at all, because it is a simple move to the next higher byte. Just like the multiplication by 10 in the decimal world is simply moving the number one left and write a zero to the least significant digit.

So let's go to a practical example. First we need some registers to

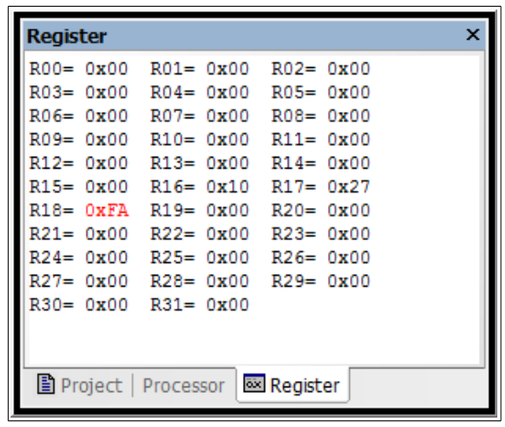
- load the numbers m1 and m2,

- provide space for the result, which might have 24 bits length.

```
;
; Test hardware multiplication 16-by-8-bit
;
; Register definitions:
;
.def Res1 = R2
.def Res2 = R3
.def Res3 = R4
.def m1L = R16
.def m1M = R17
.def m2 = R18
```

First we load the numbers:

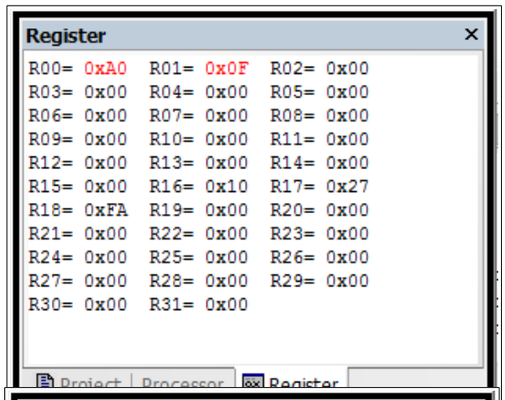
```
;
; Load Registers
;
.equ m1 = 10000
;
        ldi m1M,HIGH(m1) ; upper 8 bits of m1 to m1M
        ldi m1L,LOW(m1) ; lower 8 bits of m1 to m1L
        ldi m2,250 ; 8-bit constant to m2
```



The two numbers are loaded into R17:R16 (dec 10000 = hex 2710) and R18 (dec 250 = hex FA).

Then we multiply the LSB first:

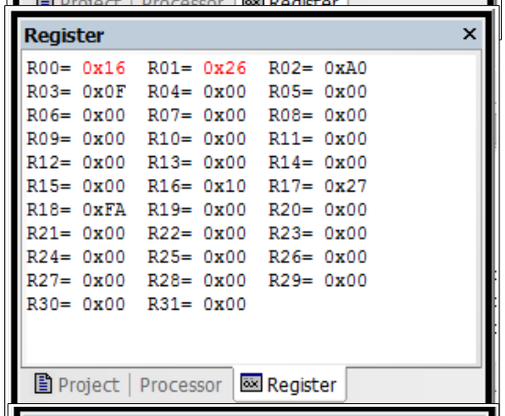
```
;
; Multiply
;
        mul m1L,m2 ; Multiply LSB
        mov Res1,R0 ; copy result to result register
        mov Res2,R1
```



The LSB multiplication of hex 27 by hex FA yields hex 0F0A, written to the registers R00 (LSB, hex A0) and R01 (MSB, hex 0F). The result is copied to the lower two bytes of the result register, R3:R2.

Now the multiplication of the MSB of m1 with m2 follows:

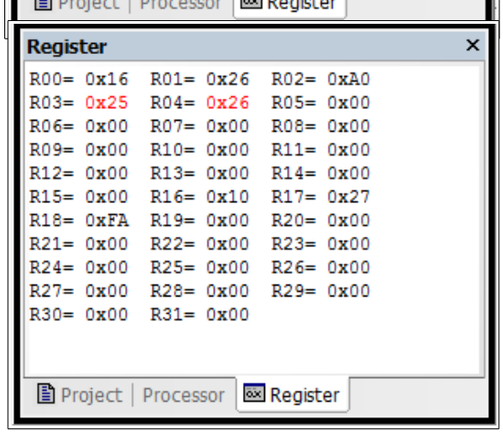
```
        mul m1M,m2 ; Multiply MSB
```



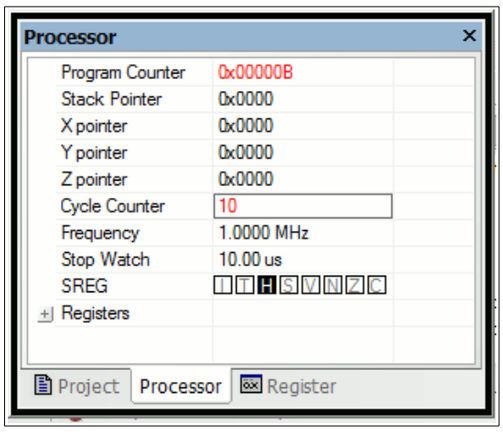
The multiplication of the MSB of m1, hex 10, with m2, hex FA, yields hex 2616 in R1:R0.

Now two steps are performed at once: multiplication by 256 and adding the result to the previous result. This is done by adding R1:R0 to Res3:Res2 instead of Res2:Res1. R1 can just be copied to Res3. R0 is added to Res2 then. If the carry is set after adding, the next higher byte Res3 is increased by one.

```
        mov Res3,R1 ; copy MSB result to result byte 3
        add Res2,R0 ; add LSB result to result byte 2
        brcc NoInc ; if not carry, jump
        inc Res3
NoInc:
```



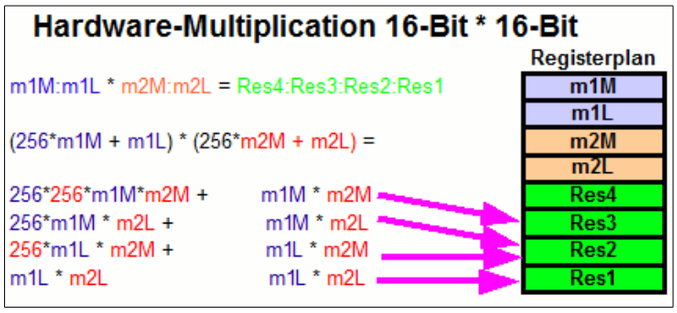
The result in R4:R3:R2 is hex 2625A0, which is decimal 2500000 (as everybody knows), and is obviously correct.



The cycle counter of the multiplication points to 10, at 1 MHz clock a total of 10 microseconds. Very much faster than software multiplication!

9.7.3 Hardware multiplication of a 16- by a 16-bit-binary

Now that we have understood the principle, it should be easy to do 16-by-16. The result requires four bytes now (Res4:Res3:Res2:Res1, located in R5:R4:R3:R2). The formula is:



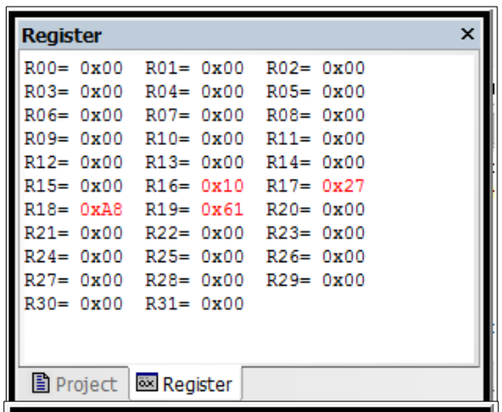
$$
\begin{aligned}
m1 * m2 &= (256*m1M + m1L) * \\
&\quad (256*m2M + m2L) \\
&= 65536*m1M*m2M + \\
&\quad 256*m1M*m2L + \\
&\quad 256*m1L*m2M + \\
&\quad m1L*m2L
\end{aligned}
$$

Obviously four multiplications now. We start with the first and the last as the two easiest ones: their results are simply copied to the correct result register positions. The results of the two multiplications in the middle of the formula have to be added to the middle of our result registers, with possible carry overflows to the most significant byte of the result. To do that, you will see a simple trick that is easy to understand. The software:

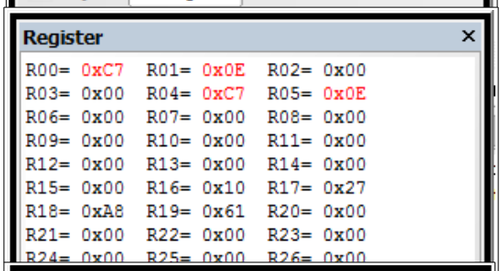
```
;
; Test Hardware Multiplication 16 by 16
;
; Define Registers
;
.def Res1 = R2
.def Res2 = R3
.def Res3 = R4
.def Res4 = R5
.def m1L = R16
.def m1M = R17
.def m2L = R18
.def m2M = R19
.def tmp = R20
;
; Load input values
;
.equ m1 = 10000
.equ m2 = 25000
;
        ldi m1M,HIGH(m1)
        ldi m1L,LOW(m1)
        ldi m2M,HIGH(m2)
        ldi m2L,LOW(m2)
;
; Multiply
;
        clr R20 ; clear for carry operations
        mul m1M,m2M ; Multiply MSBs
        mov Res3,R0 ; copy to MSW Result
        mov Res4,R1
        mul m1L,m2L ; Multiply LSBs
        mov Res1,R0 ; copy to LSW Result
        mov Res2,R1
        mul m1M,m2L ; Multiply 1M with 2L
        add Res2,R0 ; Add to Result
        adc Res3,R1
        adc Res4,tmp ; add carry
        mul m1L,m2M ; Multiply 1L with 2M
```

```
        add Res2,R0 ; Add to Result
        adc Res3,R1
        adc Res4,tmp
;
; Multiplication done
;
```

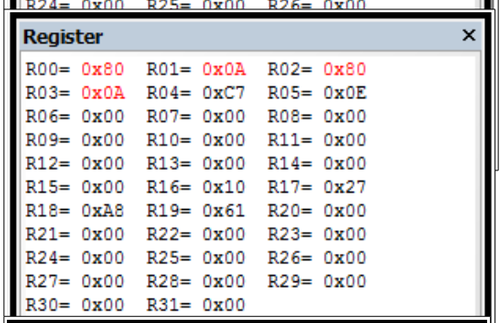
Simulation shows the following steps.



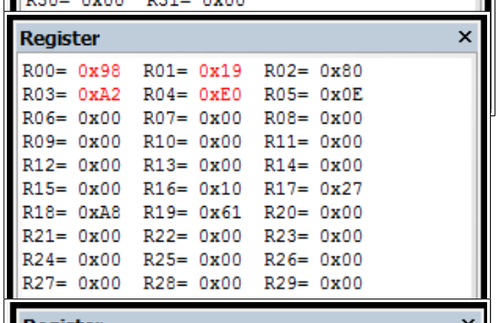
Loading the two constants 10000 (hex 2710) and 25000 (hex 61A8) to the registers in the upper register space ...



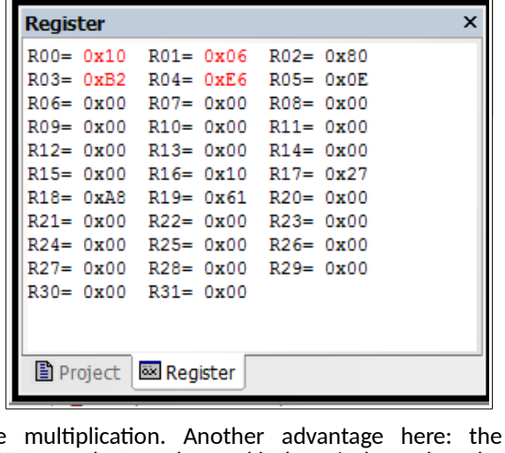
Multiplying the two MSBs (hex 27 and 61) and copying the result in R1:R0 to the two most upper result registers R5:R4 ...



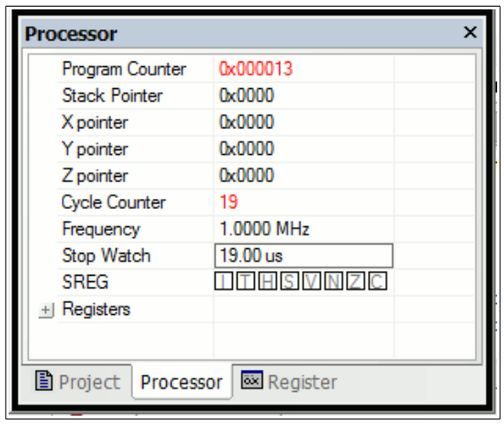
Multiplying the two LSBs (hex 10 and A8) and copying the result in R1:R0 to the two lower result registers R3:R2 ...



Multiplying the MSB of m1 with the LSB of m2 and adding the result in R1:R0 to the result register's two middle bytes, no carry occurred ...



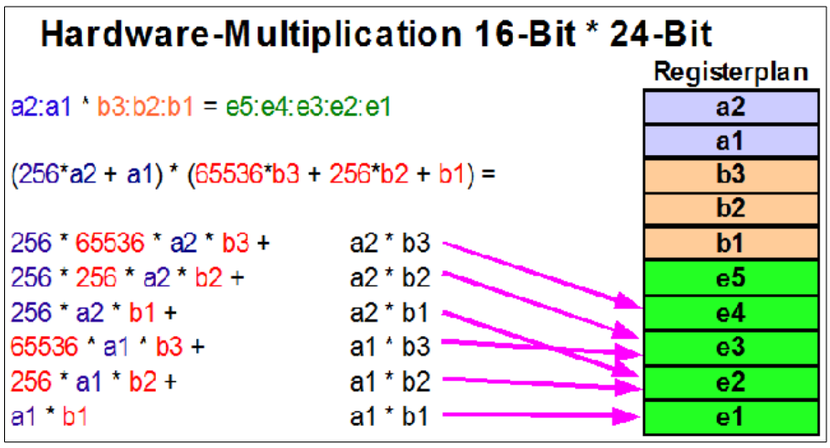
Multiplying the LSB of m1 with the MSB of m2 and adding the result in R1:R0 to the result register's two middle bytes, no carry occurred. The result is hex 0EE6B280, which is 250000000 and obviously correct ...



Multiplication needed 19 clock cycles, which is very much faster than with software multiplication. Another advantage here: the required time is ALWAYS exactly 19 cycles, and it doesn't depend on the input numbers (like is the case with software multiplication and on overflow occurrences (thanks to our small trick of adding zero with carry). So you can rely on this ...

9.7.4 Hardware multiplication of a 16- by a 24-bit-binary

The multiplication of a 16 bit binary "a" with a 24 bit binary "b" leads to results with up to 40 bit length. The multiplication scheme requires six 8-by-8-bit multiplications and adding the results to the appropriate position in the result registers.



The assembler source code for this:

```
; Hardware Multiplication 16 by 24 bit
.include "m8def.inc"
;
; Register definitions
.def a1 = R2 ; define 16-bit register
.def a2 = R3
.def b1 = R4 ; define 24-bit register
.def b2 = R5
.def b3 = R6
.def e1 = R7 ; define 40-bit result register
.def e2 = R8
.def e3 = R9
.def e4 = R10
.def e5 = R11
.def c0 = R12 ; help register for adding
.def rl = R16 ; load register
;
; Load constants
.equ a = 10000 ; multiplicator a, hex 2710
.equ b = 1000000 ; multiplicator b, hex 0F4240
        ldi rl,BYTE1(a) ; load a
        mov a1,rl
        ldi rl,BYTE2(a)
        mov a2,rl
        ldi rl,BYTE1(b) ; load b
        mov b1,rl
        ldi rl,BYTE2(b)
        mov b2,rl
        ldi rl,BYTE3(b)
        mov b3,rl
;
; Clear registers
        clr e1 ; clear result registers
        clr e2
        clr e3
        clr e4
        clr e5
        clr c0 ; clear help register
;
; Multiply
        mul a2,b3 ; term 1
        add e4,R0 ; add to result
        adc e5,R1
        mul a2,b2 ; term 2
        add e3,R0
        adc e4,R1
        adc e5,c0 ; (add possible carry)
        mul a2,b1 ; term 3
        add e2,R0
        adc e3,R1
        adc e4,c0
        adc e5,c0
        mul a1,b3 ; term 4
        add e3,R0
        adc e4,R1
        adc e5,c0
        mul a1,b2 ; term 5
        add e2,R0
        adc e3,R1
        adc e4,c0
        adc e5,c0
        mul a1,b1 ; term 6
        add e1,R0
        adc e2,R1
        adc e3,c0
        adc e4,c0
        adc e5,c0
;
; done.
        nop
; Result should be hex 02540BE400
```

The complete execution requires

- 10 clock cycles for loading the constants,
- 6 clock cycles for clearing registers, and
- 33 clock cycles for multiplication.

9.8 Division

No, unfortunately there is no hardware division. You need to do this in software!

9.8.1 Decimal division

Again we start with the decimal division, to better understand the binary division. We assume a division of 5678 by 12. This is done like this:

```
             5678 : 12 = ?
-------------------------
- 4 * 1200 = 4800
             ----
              878
- 7 *  120 =  840
              ---
               38
- 3 *   12 =   36
               --
                2
Result: 5678 : 12 = 473 Remainder 2
==================================
```

9.8.2 Binary division

In binary the multiplication of the second number in the above decimal example (4 * 1200, etc.) is not necessary, due to the fact that we have only 0 and 1 as digits. Unfortunately binary numbers have much more single digits than their decimal equivalent, so transferring the decimal division to its binary equivalent is a little bit inconvenient. So the program works a bit different than that.

The division of a 16-bit binary number by a 8-bit binary in AVR assembler is listed in the following section.

```
; Div8 divides a 16-bit-number by a 8-bit-number (Test: 16-bit-number: 0xAAAA, 8-bit-number: 0x55)
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc" ; adjust the correct path to your system!
.LIST
; Registers
.DEF rd1l = R0 ; LSB 16-bit-number to be divided
.DEF rd1h = R1 ; MSB 16-bit-number to be divided
.DEF rd1u = R2 ; interim register
.DEF rd2  = R3 ; 8-bit-number to divide with
.DEF rel  = R4 ; LSB result
.DEF reh  = R5 ; MSB result
.DEF rmp  = R16; multipurpose register for loading
;
.CSEG
.ORG 0
      rjmp start
start:
; Load the test numbers to the appropriate registers
      ldi rmp,0xAA ; 0xAAAA to be divided
      mov rd1h,rmp
      mov rd1l,rmp
      ldi rmp,0x55 ; 0x55 to be divided with
      mov rd2,rmp
; Divide rd1h:rd1l by rd2
div8:
      clr rd1u ; clear interim register
      clr reh  ; clear result (the result registers
      clr rel  ; are also used to count to 16 for the
      inc rel  ; division steps, is set to 1 at start)
; Here the division loop starts
div8a:
      clc      ; clear carry-bit
      rol rd1l ; rotate the next-upper bit of the number
      rol rd1h ; to the interim register (multiply by 2)
      rol rd1u
      brcs div8b ; a one has rolled left, so subtract
      cp rd1u,rd2 ; Division result 1 or 0?
      brcs div8c  ; jump over subtraction, if smaller
div8b:
      sub rd1u,rd2; subtract number to divide with
      sec      ; set carry-bit, result is a 1
      rjmp div8d  ; jump to shift of the result bit
div8c:
      clc      ; clear carry-bit, resulting bit is a 0
```

```
div8d:
        rol rel  ; rotate carry-bit into result registers
        rol reh
        brcc div8a  ; as long as zero rotate out of the result registers: go on with the division loop
; End of the division reached
stop:
        rjmp stop   ; endless loop
```

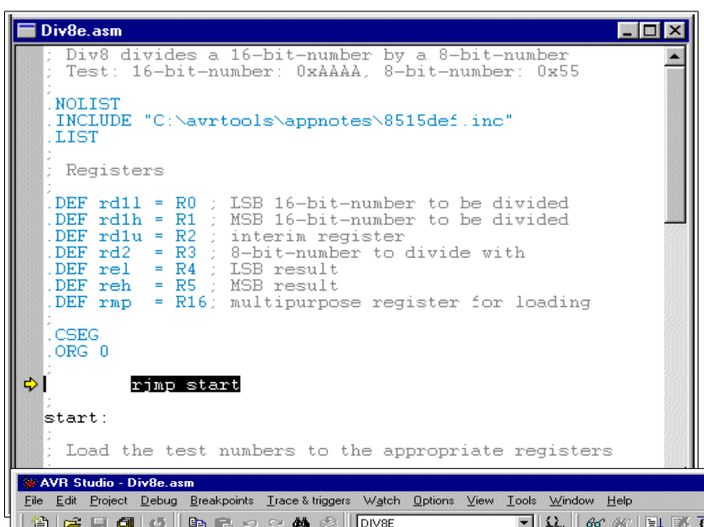
9.8.3 Program steps during division

During execution of the program the following steps are ran:

- Definition and preset of the registers with the test binaries,
- presetting the interim register and the result register pair (the result registers are presetted to 0x0001! After 16 rotations the rolling out of the one stops further division steps.),
- the 16-bit-binary in rd1h:rd1l is rotated bitwise to the interim register rd1u (multiplication by 2), if a 1 is rotated out of rd1u, the program branches to the subtraction step in step 4 immediately,
- the content of the interim register is compared with the 8-bit binary in rd2, if rd2 is smaller it is subtracted from the interim register and the carry-bit is set to one, if rd2 is greater the subtraction is skipped and a zero is set to the carry flag,
- the content of the carry flag is rotated into the result register reh:rel from the right,
- if a zero rotated out of the result register, we have to repeat the division loop, if it was a one the division is completed.

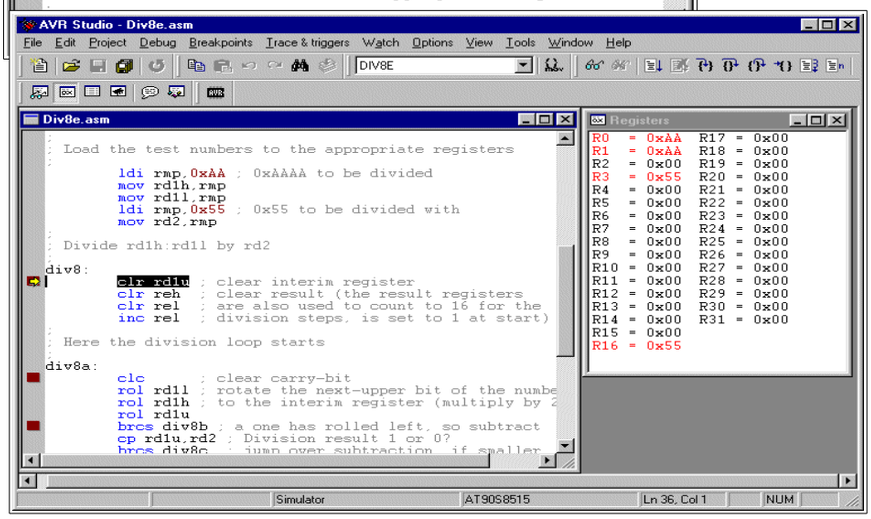
If you don't understand rotation yet you'll find this operation discussed in the multiplication section.

9.8.4 Division in the simulator



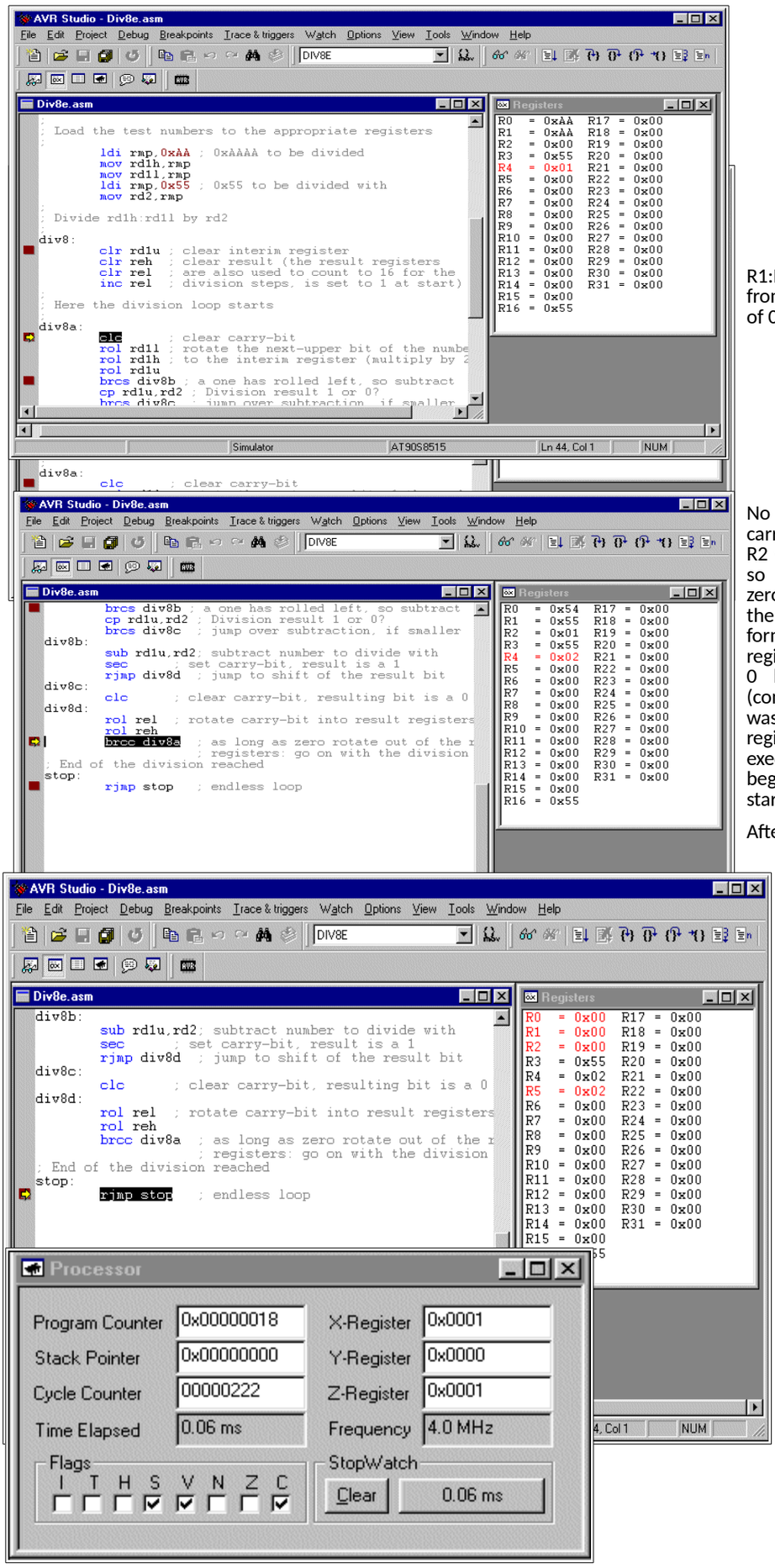
The following screen shots demonstrate the program steps in the studio (here in version 3, so it looks different). To do this, you have to assemble the source code and open the resulting object file in the studio.

The object code has been started, the cursor (yellow arrow) is on the first executable instruction. The key F11 performs single steps.



The test binaries 0xAAAA and 0x55, to be divided, have been written to the registers R1:R0 and R3.

The interim register R2 and the result register pair are set to their predefined values.



R1:R0 was rotated left to R2, from 0xAAAA the doubled value of 0x015554 was yielded.

No overflow from rotation into carry has occurred and 0x01 in R2 was smaller than 0x55 in R3, so subtraction was skipped. A zero in the carry is rotated into the result register R5:R4. The former content of the result register, a single 1-bit in position 0 has rotated to position 1 (content now: 0x0002). As a zero was rotated out of the result register pair, the next step to be executed is a branch to the beginning of the division loop start and the loop is repeated.

After executing the loop 16 times we have reached the breakpoint set at the end of the division routine. The result register in R5:R4 holds 0x0202, the result of the division. The registers R2:R1:R0 are empty, so we do not have a remainder left. If a remainder would have been resulted we can use it to decide whether an incrementation of the result should take place, rounding of the result up. This step is not coded here.

The whole division needs 60 micro-seconds processor time (open a processor view in the studio menu). A rather long time for a division.

9.9 Number conversion

Number conversion routines are not included here. Please refer to the website at

http://www.avr-asm-tutorial.net/avr_en

if you need the source code or a better understanding.

9.10 Decimal Fractions

First: Do not use any floating points, unless you really need them. Floating points are resource killers in an AVR, lame ducks and need extreme execution times. Run into this dilemma, if you think assembler is too complicated, and you prefer Basic or other languages like C or Pascal.

Not so, if you use assembler. You'll be shown here, how you can perform the multiplication of a fixed point real number in less than 60 micro-seconds, in special cases even within 18 micro-seconds, at 4 MHz clock frequency. Without any floating point processor extensions and other expensive tricks for people too lazy to use their brain.

How to do that? Back to the roots of math! Most tasks with floating point reals can be done using integer numbers. Integers are easy to program in assembler and perform fast. The decimal point is only in the brain of the programmer, and is added somewhere in the decimal digit stream. No one realizes, that this is a trick.

9.10.1 Linear conversions

As an example the following task: an 8-Bit-AD-Converter measures an input signal in the range from 0.00 to 2.55 Volt, and returns as the result a binary in the range from $00 and $FF. The result, a voltage, is to be displayed on a LCD display. Silly example, as it is so easy: The binary is converted to a decimal ASCII string between 000 and 255, and just behind the first digit the decimal point has to be inserted. Done!

The electronics world sometimes is more complicated. E. g., the AD-Converter returns an 8-Bit-Hex for input voltages between 0.00 and 5.00 Volt. Now we're tricked and do not know how to proceed. To display the correct result on the LCD we would have to multiply the binary by 500/255, which is 1.9608. This is a silly number, as it is almost 2, but only almost. And we don't want that kind of inaccuracy of 2%, while we have an AD-converter with around 0.25% accuracy.

To cope with this, we multiply the input by 500/255*256 or 501.96 and divide the result by 256. Why first multiply by 256 and then divide by 256? It's just for enhanced accuracy. If we multiply the input by 502 instead of 501.96, the error is just in the order of 0.008%. That is good enough for our AD-converter, we can live with that. And dividing by 256 is an easy task, because it is a well-known power of 2. By dividing with numbers that are a power of 2, the AVR feels very comfortable and performs very fast. By dividing with 256, the AVR is even faster, because we just have to skip the last byte of the binary number. Not even shift and rotate!

The multiplication of an 8-bit-binary with the 9-bit-binary 502 (hex 1F6) can have a result larger than 16 bits. So we have to reserve 24 bits or 3 registers for the result. During multiplication, the constant 502 has to be shifted left (multiplication by 2) to add these numbers to the result each time a one rolls out of the shifted input number. As this might need eight shifts left, we need further three bytes for this constant. So we chose the following combination of registers for the multiplication:

Number	*Value (example)*	*Register*
Input value	255	R1
Multiplicand	502	R4 : R3 : R2
Result	128,010	R7 : R6 : R5

After filling the value 502 (00.01.F6) to R4 : R3 : R2 and clearing the result registers R7 : R6 : R5, the multiplication goes like this:

1. Test, if the input number is already zero. If yes, we're done.
2. If no, one bit of the input number is shifted out of the register to the right, into the carry, while a zero is stuffed into bit 7. This instruction is named Logical-Shift-Right or LSR.
3. If the bit in carry is a one, we add the multiplicand (during step 1 the value 502, in step 2 it's 1004, a. s. o.) to the result. During adding, we care for any carry (adding R2 to R5 by ADD, adding R3 to R6 and R4 to R7 with the ADC instruction!). If the bit in the carry was a zero, we just don't add the multiplicand to the result and jump to the next step.
4. Now the multiplicand is multiplied by 2, because the next bit shifted out of the input number is worth double as much. So we shift R2 to the left (by inserting a zero in bit 0) using LSL. Bit 7 is shifted to the carry. Then we rotate this carry into R3, rotating its content left one bit, and bit 7 to the carry. The same with R4.
5. Now we're done with one digit of the input number, and we proceed with step 1 again.

The result of the multiplication by 502 now is in the result registers R7 : R6 : R5. If we just ignore register R5 (division by 256), we have our desired result. To enhance accuracy, we can use bit 7 in R5 to round the result. Now we just have to convert the result from its binary form to decimal ASCII (see Conversion bin to decimal-ASCII on the website). If we just add a decimal point in the right place in the ASCII string, our voltage string is ready for the display.

The whole program, from the input number to the resulting ASCII string, requires between 79 and 228 clock cycles, depending from the input number. Those who want to beat this with the floating point routine of a more sophisticated language than assembler, feel free to mail me your conversion time (and program flash and memory usage).

9.10.2 Example 1: 8-bit-AD-converter with fixed decimal output

```
; Demonstrates floating point conversion in Assembler, (C)2003 www.avr-asm-tutorial.net
;
; The task: You read in an 8-bit result of an analogue-digital-converter, number is in the range from hex 00 to FF.
;    You need to convert this into a floating point number in the range from 0.00 to 5.00 Volt
; The program scheme:
;    1. Multiplication by 502 (hex 01F6).That step multiplies by 500, 256 and divides by 255 in one step!
;    2. Round the result and cut the last byte of the result. This step divides by 256 by ignoring the last byte of the result.
;       Before doing that, bit 7 is used to round the result.
;    3. Convert the resulting word to ASCII and set the correct decimal sign. The resulting word in the range from 0 to 500
```

```
;       is displayed in ASCII-characters as 0.00 to 5.00.
; The registers used:
;    The routines use the registers R8..R1 without saving these before. Also required is a multipurpose register called rmp,
;    located in the upper half of the registers. Please take care that these registers don't conflict with the register use in the
;    rest of your program.
;    When entering the routine the 8-bit number is expected in the register R1. The multiplication uses R4:R3:R2 to hold
;    the multiplicator 502 (is shifted left max. eight times during multiplication). The result of the multiplication is calculated
;    in the registers R7:R6:R5. The result of the so called division by 256 by just ignoring R5 in the result, is in R7:R6. R7:R6
;    is rounded, depending on the highest bit of R5, and the result is copied to R2:R1.
;    Conversion to an ASCII-string uses the input in R2:R1, the register pair R4:R3 as a divisor for conversion, and places the
;    ASCII result string to R5:R6:R7:R8 (R6 is the decimal char).
; Other conventions:
;   The conversion uses subroutines and the stack.The stack must work fine for the use of three levels (six bytes SRAM).
; Conversion times:
;   The whole routine requires 228 clock cycles  maximum (converting $FF), and 79 clock cycles minimum (converting $00).
;    At 4 MHz the times are 56.75 microseconds resp. 17.75 microseconds.
; Definitions:
;   Registers
.DEF rmp = R16 ; used as multi-purpose register
;   AVR type: Tested for type AT90S8515, only required for stack setting, routines work fine with other AT90S-types also
.NOLIST
.INCLUDE "8515def.inc"
.LIST
; Start of  test program
; Just writes a number to R1 and starts the conversion routine, for test purposes only
.CSEG
.ORG $0000
        rjmp main
main:
        ldi rmp,HIGH(RAMEND) ; Set the stack
        out SPH,rmp
        ldi rmp,LOW(RAMEND)
        out SPL,rmp
        ldi rmp,$FF ; Convert $FF
        mov R1,rmp
        rcall fpconv8 ; call the conversion routine
no_end:   ; unlimited loop, when done
        rjmp no_end
; Conversion routine wrapper, calls the different conversion steps
fpconv8:
        rcall fpconv8m ; multiplicate by 502
        rcall fpconv8r ; round and divide by 256
        rcall fpconv8a ; convert to ASCII string
        ldi rmp,'.' ; set decimal char
        mov R6,rmp
        ret ; all done
; Subroutine multiplication by 502
fpconv8m:
        clr R4 ; set the multiplicant to 502
        ldi rmp,$01
        mov R3,rmp
        ldi rmp,$F6
        mov R2,rmp
        clr R7 ; clear the result
        clr R6
        clr R5
fpconv8m1:
        or R1,R1 ; check if the number is all zeros
        brne fpconv8m2 ; still one's, go on convert
        ret ; ready, return back
fpconv8m2:
        lsr R1 ; shift number to the right (div by 2)
        brcc fpconv8m3 ; if the lowest bit was 0, then skip adding
        add R5,R2 ; add the number in R6:R5:R4:R3 to the result
        adc R6,R3
        adc R7,R4
fpconv8m3:
        lsl R2 ; multiply R4:R3:R2 by 2
        rol R3
        rol R4
        rjmp fpconv8m1 ; repeat for next bit
; Round the value in R7:R6 with the value in bit 7 of R5
fpconv8r:
        clr rmp ; put zero to rmp
        lsl R5 ; rotate bit 7 to carry
        adc R6,rmp ; add LSB with carry
        adc R7,rmp ; add MSB with carry
        mov R2,R7 ; copy the value to R2:R1 (divide by 256)
        mov R1,R6
        ret
```

```
; Convert the word in R2:R1 to an ASCII string in R5:R6:R7:R8
fpconv8a:
	clr R4 ; Set the decimal divider value to 100
	ldi rmp,100
	mov R3,rmp
	rcall fpconv8d ; get ASCII digit by repeated subtraction
	mov R5,rmp ; set hundreds string char
	ldi rmp,10 ; Set the decimal divider value to 10
	mov R3,rmp
	rcall fpconv8d ; get the next ASCII digit
	mov R7,rmp ; set tens string char
	ldi rmp,'0' ; convert the rest to an ASCII char
	add rmp,R1
	mov R8,rmp ; set ones string char
	ret
; Convert binary word in R2:R1 to a decimal digit by substracting the decimal divider value in R4:R3 (100, 10)
fpconv8d:
	ldi rmp,'0' ; start with decimal value 0
fpconv8d1:
	cp R1,R3 ; Compare word with decimal divider value
	cpc R2,R4
	brcc fpconv8d2 ; Carry clear, subtract divider value
	ret ; done subtraction
fpconv8d2:
	sub R1,R3 ; subtract divider value
	sbc R2,R4
	inc rmp ; up one digit
	rjmp fpconv8d1 ; once again
; End of conversion test routine
```

9.10.3 Example 2: 10-bit-AD-converter with fixed decimal output

This example is a bit more complicated. Refer to the website if you need it.

10 Project planning

10.1 How to plan an AVR project in assembler

Here are the basics on how to plan a simple project, to be programmed in assembler. Because the hardware components determine a lot, the hardware considerations are discussed first. Then a chapter on interrupt follows and after that timing issues are discussed.

10.2 Hardware considerations

The decision, which type of AVR fits best to your needs, a number of considerations can play a role. Here are the most relevant ones:

1. Which port connections with a fixed location are needed? Fixed locations are I/O ports of internal components that are only available on certain pins, and cannot be moved to another portpin. Components and connections of this kind are:
    1. If the processor should be programmable within the circuit (ISP interface), the pins SCK, MOSI and MISO have to be assigned for this purpose. If your peripheral allows that those can be used as inputs (e. g. SCK and MOSI) or as outputs (by decoupling them via resistors or multiplexers).
    2. If a serial interface is needed, RXD and TXD have to be reserved for that purpose. If the RTS/CTS hardware handshake protocol shall be implemented as well, two additional portpins are required, but can be placed at any other free location.
    3. If an analog comparator is needed, AIN0 and AIN1 have to be reserved for that.
    4. If external signals are to be monitored for level changes, INT0 and/or INT1 have to be reserved for that.
    5. If AD converters arew to be used, the ADC inputs have to be placed and reserved for that purpose. If the converter has the external AVCC and AREF connections, those should be used and wired accordingly.
    6. If external pulses are to be counted, the timer input pins T0, T1 and T2 are fixed and exclusive for that use.
    7. If external SRAM is to be attached, the respective address and data ports together with ALE, RD and WR have to be reserved.
    8. If the processor clock should be generated from an external crystal oscillator, XTAL1 has to be reserved. If an external crystal or ceramic resonator shall control the clock frequency, XTAL1 and XTAL2 are fixed for that purpose.
2. Are there any external components that require more than one portpin (e. g. 2, 4 or 8) to be written or read? These should be defined in an appropriate way (in the same port, in the right order).
    1. If controlling of an external device requires writing or reading of more than one bit at once, e. g. a four- or eight-bit LCD interface, the necessary port bits should be in the right order. If it isn't possible to place the whole interface in a single port, the interface can be divided into two pieces. The software is easier, if the resulting portions are left- or right-adjusted in the port.
    2. If two or more ADC channels are required, the software is easier, if those are placed in an order (e. g. ADC2+ADC3+ADC4).
3. At the end, all external components are placed that do not require fixed pins.
    1. If only a single pin causes you to select a larger device, you can consider using the RESET pin for that purpose. This can be used as an input pin if a certain fuse is set. Setting of that fuse disables further ISP programming, the chip can only be programmed in high-voltage programming modes. For final productions with a large number of identical devices, this is acceptable, but not for prototyping. In case of prototyping a high-voltage programmer interface on the ISP pin can be used, if the component on the RESET pin is protected against the 12 V on this pin during ISP programming, e. g. with a resistor and a zener diode.

Further considerations for the decision, which processor type fits best, are:

- How many timers are needed and which resolution should these provide?
- Which values/informations should be preserved when the operating voltage is shut down? (EEPROM capacity)
- How much storage space is required? (SRAM capacity)
- Space requirements on your PCB, how much space for the processor fits best, which package types are available?
- Operating voltages and power requirements. If the operating voltage comes from a battery or an accumulator, the power characteristics play an important role.
- Price for the device? Only relevant for production in larger series. Not at all depending from the processor's internals, and a matter of unpredictable market conditions.
- Availability? If one starts a project with the AT90S1200, probably from a rummage table, can make it cheap. Such a decision is not very sustainable. Porting such a project to a tiny- or mega-device mostly ends up in a complete redesign of the software, that also looks and feels much better, works better and requires only a fraction of code lines.

10.3 Considerations on interrupt operation

Very simple tasks work fine without interrupts. If power consumption is an issue, this ain't true either. Nearly all projects require interrupts. And this should be planned thoroughly.

10.3.1 Basic requirements of interrupt-driven operation

If not aware any more, here are the basics.

- Enabling interrupts:
    - Interrupts require the stack hardware. So set SPL (and in larger devices SPH) at the beginning to RAMEND, and reserve the upper part of the SRAM for that purpose (e. g. the last 8 to x bytes).
    - Each internal component and each condition that should be able to trigger an interrupt has to be enabled to do that, by setting the respective interrupt enable flag bit. Switching those bits on and off is a risky thing, so better design your software without switching.
    - The I flag in the status register SREG has to be set at the begin and remains set during operation. If it is necessary to clear the I flag during an operation outside an interrupt service routine, add the set-I-flag

instruction within a few instruction words.
- Interrupt vector table:
  - Each internal component and each enabled interrupt condition corresponds to a specific interrupt vector, placed in a certain address in the flash program storage. The instruction at this address is a single-word RJMP instruction, in large ATmega processors a two-word JMP instruction, to the respective interrupt service routine.
  - The vector addresses are type-specific. When porting the software to a different type, those require adjustment.
  - Each vector address in the table, that is currently not used, is given a RETI instruction (in large ATmega types a RETI, followed by an NOP). That prevents erroneous ghost interrupts to be running into false code. The use of the .ORG directive for adjusting vector addresses does not provide safety against those events.
  - If an interrupt condition occurs, the respective flag in the control register of the internal component is set. This is automatically cleared, if the interrupt is executed. In some rare cases (e. g. in case of a TX buffer empty interrupt condition of an UART, if no further character is to be sent), the interrupt enable flag of the component has to be cleared first and the interrupt condition flag at last.
  - If the interrupt condition gets true for more than one component at a time, the interrupt with a lower address wins the race.
- Interrupt service routines:
  - Each service routine starts with saving the status register SREG in a register exclusively reserved for that purpose, and ends with restoring that status register. Because interrupts can occur every time, also in times while the processor is performing instructions in the main program loop, any disturbance of that status register can cause unpredictable malfunctions.
  - Before jumping to the service routine, the processor pushes the current instruction counter to the stack. The interrupt and the jump to the respective service routine disables further interrupts by temporarily clearing the I flag in the status register SREG. Each service routine ends with the instruction RETI, that pops the instruction pointer from the stack and sets the I flag on again.
  - Because the execution of an interrupt service routine blocks any further interrupt requests from being served, even those of a higher priority, each service routine has to be as short as possible and performs only the time critical portions of the task. Lengthy response operations have to be performed outside the interrupt service routine.
  - Because interrupting an interrupt service routine does not happen, all interrupt service routines can use the same temporary register.
- Interfacing interrupt service routine and main program loop:
  - The communication between the interrupt serve routine and the main program loop is performed via single flags, that are set within the service routine and cleared in the main program loop. The clearing of flags is either performed in single word instructions or interrupts are temporarily disabled during that step to block erroneous overwriting of other flags that were possibly changed in between the three steps read-modify-write.
  - Values that the interrupt service routines provides are handed over in dedicated registers or in specific SRAM locations. Each change of those values, that are used later on outside the service routine, have to be checked for possible corruption, if another interrupt can occur in between. Single byte handling is easy, but handing over two or more bytes requires a hand-over protocol (interrupt disable during hand-over, flag setting to prevent overwrite, etc.). As an example, the handover of a 16-bit timer value requires disabling of interrupts first. Otherwise the first byte read does not necessarily correspond to the second byte read, if another interrupt happened in between.
- Main program loop:
  - Within the main program loop the processor is sent to sleep, with the sleep mode "idle" selected. Each interrupt wakes up the processor, jumps to the respective interrupt service routine and, after its return from interrupt, continues its operation in the main program loop. It makes sense to check for any flags that were set within the service routine. If that is the case, the treatment of the flag can be performed. After all things were finalized, another check for the flag settings can be made (in case of long routines) and the processor can be sent back to sleep.

10.3.2 Example for an interrupt-driven assembler program

The following provides an example for an interrupt-driven assembler program, that utilizes all the above mentioned rules.

```
;
; Register definitions
;
.EQU rsreg = R15 ; saving the status during interrupts
.EQU rmp = R16 ; Temporary register outside interrupts
.EQU rimp = R17 ; Temporary register inside interrupts
.EQU rflg = R18 ; Flag register for communication
.EQU bint0 = 0 ; Flag bit for signaling INT0-Service
.EQU btc0 = 1 ; Flag bit for signaling TC0-Overflow
; ...
; ISR-Table
;
.CSEG
.ORG $0000
        rjmp main ; Reset vector, executed at start-up
        rjmp isr_int0 ; INT0-vector, executed on level changes on the INT0 input
line
        reti ; unused interrupt
        reti ; unused interrupt
        rjmp isr_tc0_Overflow ; TC0-Overflow-vector, executed in case of a TC0
overflow
        reti ; unused interrupt
        reti ; unused interrupt
```

```
        ; ... other int vectors
;
; Interrupt service routines
;
isr_int0: ; INT0-Service Routine
        in rsreg,SREG ; safe status
        in rimp,PINB ; read port B to temp register
        out PORTC,rimp ; write temp register to port C
        ; ... do other things
        sbr rflg,1<<bint0 ; signaling INT0 to outside
out SREG,rsreg ; restore status
     reti ; return back and enable interrupts
isr_tc0_Overflow: ; TC0 Overflow Service Routine
     in rsreg,SREG ; safe status
     in rimp,PINB ; read port B in temp register
     out PORTC,rimp ; write temp register to port C
     ; ... do other things
     sbr rflg,1<<btc0 ; set TC0-flag
     out SREG,rsreg ; restore status
     reti ; return back and enable interrupts
;
; Main program start
;
main:
     ldi rmp,HIGH(RAMEND) ; set stack register
     out SPH,rmp
     ldi rmp,LOW(RAMEND)
     out SPL,rmp
     ; ... other things to do
     ; INT Enable for TC0 overflows
     ldi rmp,1<<TOIE0 ; Overflow Interrupt Enable Timer 0
     out TIMSK,rmp ; set interrupt-mask of the timer
     ldi rmp,(1<<CS00)|(1<<CS02) ; prescaler by 1024
     out TCCR0,rmp ; start timer
     ; INT Enable of the INT0 input
     ldi rmp,(1<<SE)|(1<<ISC00) ; SLEEP-Enable and INT0 int on all level changes
     out MCUCR,rmp ; to the control register
     ldi rmp,1<<INT0 ; enable INT0 interrupts
     out GICR,rmp ; to the interrupt control register
     ; set interrupt status flag
     sei ; set interrupt flag
;
; Main program loop
;
loop:
     sleep ; processor to sleep
     nop ; dummy for wake-up
     sbrc rflg,bint0 ; INT0 flag not set
     rcall mache_int0 ; handle INT0 event
     sbrc rflg,btc0 ; TC0-Overflow flag not set
     rcall mache_tc0 ; handle TC0 overflow
     rjmp loop ; go back to sleep
;
; Handle event results
;
mache_int0: ; handle INT0 result
     cbr rflg,1<<bint0 ; clear INT0 flag
     ; ... do other things
     ret ; ready, back to loop
mache_tc0: ; handle TC0 overflow
     cbr rflg,1<<btc0 ; clear TC0 flag
     ; ... do other things
     ret ; ready, back to loop
```

10.4 Considerations on timing

If an AVR project goes beyond polling an I/O port and, depending from that result, doing something, considerations on timing are necessary. Timing

- starts with the selection of the processor type,
- continues with the question, what has be executed periodically and with which precision,
- and which timing control opportunities exist,
- how those things can be combined.

Selection of the clock frequency of the processor

The main question is on the necessary precision of the processor clock.

Is it unnecessary in the application to perform times less than a few percent inaccurate, the internal RC oscillator of most of the AVR types is sufficient. In the tiny and mega types, a oscillator calibration is built in, so that differences between the nominal and the effective frequency are reduced. Note that the default internal calibration byte was selected at a certain operating voltage. If your operating voltage is fixed at a different level, rewriting the calibration byte brings more accuracy. If the operating voltage is fluctuating, the error can be too large.

If the internal RC clock is too slow or too large, some device types have a clock prescaler on board. This feature allows to optimize the clock frequency, and different clock frequencies can be selected. This is either done once with changing a hardware fuse (the DIV8 fuse) or within the software (e. g. to reduce the supply power during pauses). But be aware that some devices with a limited clock specification (V types) should not be set to beyond their limit, otherwise they won't work correct any more.

If the internal RC oscillator is too inaccurate, fuses can be set for external RC combination, an external oscillator, a crystal (Xtal) or a ceramic device. Because false fuse setting can cause a catastophe, a rescue board with an external oscillator might be the last chance to get the device working and the fuse resetted again.

The absolute clock frequency should be appropriate for the application. As an indicator, the repeat frequency of the most often performed work package can be used. If a key has to be polled any 2 ms, and, after 20 times, should be debounced long enough, there is plenty of time, if a 1 MHz clock is used (2,000 clock cycles between any two polls, 20,000 clocks for repeated execution of the key command).

It's only getting narrower, if a pulse width modulated signal with high resolution and a high PWM frequency has to be reached. With a PWM frequency of 10 kHz and 8 bits resolution 2.56 MHz are too slow for a software-driven solution. If a timer with some software-overhead can take over that burden, that's better.

11 Annex

11.1 Instructions sorted by function

For the abbreviations used see the list of abbreviations.

Function	*Sub function*	*instruction*	*Flags*	*Clk*
Register set	0	CLR r1	Z N V	1
	255	SER rh		1
	Constant	LDI rh,c255		1
Copy	Register => Register	MOV r1,r2		1
	SRAM => Register, direct	LDS r1,c65535		2
	SRAM => Register	LD r1,rp		2
	SRAM => Register and INC	LD r1,rp+		2
	DEC, SRAM => Register	LD r1,-rp		2
	SRAM, displaced => Register	LDD r1,ry+k63		2
	Port => Register	IN r1,p1		1
	Stack => Register	POP r1		2
	Program storage Z => R0	LPM		3
	Register => SRAM, direct	STS c65535,r1		2
	Register => SRAM	ST rp,r1		2
	Register => SRAM and INC	ST rp+,r1		2
	DEC, Register => SRAM	ST -rp,r1		2
	Register => SRAM, displaced	STD ry+k63,r1		2
	Register => Port	OUT p1,r1		1
	Register => Stack	PUSH r1		2
Add	8 Bit, +1	INC r1	Z N V	1
	8 Bit	ADD r1,r2	Z C N V H	1
	8 Bit + Carry	ADC r1,r2	Z C N V H	1
	16 Bit, constant	ADIW rd,k63	Z C N V S	2
Subtract	8 Bit, -1	DEC r1	Z N V	1
	8 Bit	SUB r1,r2	Z C N V H	1
	8 Bit, constant	SUBI rh,c255	Z C N V H	1
	8 Bit - Carry	SBC r1,r2	Z C N V H	1
	8 Bit - Carry, constant	SBCI rh,c255	Z C N V H	1
	16 Bit	SBIW rd,k63	Z C N V S	2
Shift	logic, left	LSL r1	Z C N V	1
	logic, right	LSR r1	Z C N V	1
	Rotate, left over Carry	ROL r1	Z C N V	1
	Rotate, right over Carry	ROR r1	Z C N V	1
	Arithmetic, right	ASR r1	Z C N V	1
	Nibble exchange	SWAP r1		1
Binary	And	AND r1,r2	Z N V	1
	And, constant	ANDI rh,c255	Z N V	1
	Or	OR r1,r2	Z N V	1
	Or, constant	ORI rh,c255	Z N V	1
	Exclusive-Or	EOR r1,r2	Z N V	1
	Ones-complement	COM r1	Z C N V	1
	Twos-complement	NEG r1	Z C N V H	1
Bits change	Register, set	SBR rh,c255	Z N V	1
	Register, clear	CBR rh,255	Z N V	1
	Register, copy to T-Flag	BST r1,b7	T	1
	Register, copy from T-Flag	BLD r1,b7		1
	Port, set	SBI pl,b7		2
	Port, clear	CBI pl,b7		2

Function	*Sub function*	*instruction*	*Flags*	*Clk*
Status bit set	Zero-Flag	SEZ	Z	1
	Carry Flag	SEC	C	1
	Negative Flag	SEN	N	1
	Twos complement carry Flag	SEV	V	1
	Half carry Flag	SEH	H	1
	Signed Flag	SES	S	1
	Transfer Flag	SET	T	1
	Interrupt Enable Flag	SEI	I	1
Status bit clear	Zero-Flag	CLZ	Z	1
	Carry Flag	CLC	C	1
	Negative Flag	CLN	N	1
	Twos complement carry Flag	CLV	V	1
	Half carry Flag	CLH	H	1
	Signed Flag	CLS	S	1
	Transfer Flag	CLT	T	1
	Interrupt Enable Flag	CLI	I	1
Compare	Register, Register	CP r1,r2	Z C N V H	1
	Register, Register + Carry	CPC r1,r2	Z C N V H	1
	Register, constant	CPI rh,c255	Z C N V H	1
	Register, ≤0	TST r1	Z N V	1
Immediate Jump	Relative	RJMP c4096		2
	Indirect, Address in Z	IJMP		2
	Subroutine, relative	RCALL c4096		3
	Subroutine, Address in Z	ICALL		3
	Return from Subroutine	RET		4
	Return from Interrupt	RETI	I	4
Conditional Jump	Status bit set	BRBS b7,c127		1/2
	Status bit clear	BRBC b7,c127		1/2
	Jump if equal	BREQ c127		1/2
	Jump if not equal	BRNE c127		1/2
	Jump if carry set	BRCS c127		1/2
	Jump if carry clear	BRCC c127		1/2
	Jump if equal or greater	BRSH c127		1/2
	Jump if lower	BRLO c127		1/2
	Jump if negative	BRMI c127		1/2
	Jump if positive	BRPL c127		1/2
	Jump if greater or equal (Signed)	BRGE c127		1/2
	Jump if lower than zero (Signed)	BRLT c127		1/2
	Jump on half carry set	BRHS c127		1/2
	Jump if half carry clear	BRHC c127		1/2
	Jump if T-Flag set	BRTS c127		1/2
	Jump if T-Flag clear	BRTC c127		1/2
	Jump if Twos complement carry set	BRVS c127		1/2
	Jump if Twos complement carry clear	BRVC c127		1/2
	Jump if Interrupts enabled	BRIE c127		1/2
	Jump if Interrupts disabled	BRID c127		1/2
Conditioned Jumps	Register bit=0	SBRC r1,b7		1/2/3
	Register bit=1	SBRS r1,b7		1/2/3
	Port bit=0	SBIC pl,b7		1/2/3
	Port bit=1	SBIS pl,b7		1/2/3
	Compare, jump if equal	CPSE r1,r2		1/2/3
Others	No Operation	NOP		1
	Sleep	SLEEP		1
	Watchdog Reset	WDR		1

11.2 Directives and Instruction lists in alphabetic order

11.2.1 Assembler directives in alphabetic order

Directive	***... means ...***
.CSEG	*Assemble to the Code segment*
.DB	*Insert data byte(s)*
.DEF	*Define a register name*
.DW	*Insert data word(s)*
.ENDMACRO	*Macro is complete, stop recording*
.ESEG	*Assemble to the EEPROM segment*
.EQU	*Define a constant by name and set its value*
.INCLUDE	*Insert a file's content at this place as if it would be part of this file*
.MACRO	*Start to record the following instructions as a macro definition*
.ORG	*Set the assembler output address to the following number*

11.2.2 Instructions in alphabetic order

Instruction	*... performs ...*
ADC r1,r2	*Add r2 with Carry to r1 and store result in r1*
ADD r1,r2	*Add r2 to r1 and store result in r1*
ADIW rd,k63	*Add the immediate word constant k63 to the double register rd+1:rd (rd = R24, R26, R28, R30)*
AND r1,r2	*And bit wise r1 with the value in r2 and store the result in r1*
ANDI rh,c255	*And bit wise the upper register rh with the constant c255 and store the result in rh*
ASR r1	*Arithmetic shift the register r1 right*
BLD r1,b7	*Copy the T-flag in the status register to bit b7 in register r1*
BRCC c127	*Branch by c127 instructions for- or backwards if the carry flag in the status register is clear*
BRCS c127	*Branch by c127 instructions for- or backwards if the carry flag in the status register is set*
BREQ c127	*Branch by c127 instructions for- or backwards if the zero flag in the status register is set*
BRGE c127	*Branch by c127 instructions for- or backwards if the carry flag in the status register is clear*
BRHC c127	*Branch by c127 instructions for- or backwards if the half carry flag in the status register is clear*
BRHS c127	*Branch by c127 instructions for- or backwards if the half carry flag in the status register is set*
BRID c127	*Branch by c127 instructions for- or backwards if the interrupt flag in the status register is clear*
BRIE c127	*Branch by c127 instructions for- or backwards if the interrupt flag in the status register is set*
BRLO c127	*Branch by c127 instructions for- or backwards if the carry flag in the status register is set*
BRLT c127	*Branch by c127 instructions for- or backwards if the negative and overflow flag in the status register are set*
BRMI c127	*Branch by c127 instructions for- or backwards if the negative flag in the status register is set*
BRNE c127	*Branch by c127 instructions for or backwards if the zero flag in the status register is set*
BRPL c127	*Branch by c127 instructions for- or backwards if the negative flag in the status register is clear*
BRSH c127	*Branch by c127 instructions for- or backwards if the carry flag in the status register is clear*
BRTC c127	*Branch by c127 instructions for- or backwards if the transfer flag in the status register is clear*
BRTS c127	*Branch by c127 instructions for- or backwards if the transfer flag in the status register is set*
BRVC c127	*Branch by c127 instructions for- or backwards if the overflow flag in the status register is clear*
BRVS c127	*Branch by c127 instructions for- or backwards if the overflow flag in the status register is set*
BST r1,b7	*Copy the bit b7 in register r1 to the transfer flag in the status register*
CBI pl,b7	*Clear bit b7 in the lower port pl*
CBR rh,k255	*Clear all the bits in the upper register rh, that are set in the constant k255 (mask)*
CLC	*Clear the carry bit in the status register*
CLH	*Clear the half carry bit in the status register*
CLI	*Clear the interrupt bit in the status register, disable interrupt execution*
CLN	*Clear the negative bit in the status register*
CLR r1	*Clear the register r1*
CLS	*Clear the signed flag in the status register*
CLT	*Clear the transfer flag in the status register*
CLV	*Clear the overflow flag in the status register*
CLZ	*Clear the zero flag in the status register*
COM r1	*Complement register r1 (ones complement)*
CP r1,r2	*Compare register r1 with register r2*
CPC r1,r2	*Compare register r1 with register r2 and the carry flag*
CPI rh,c255	*Compare the upper register rh with the immediate constant c255*
CPSE r1,r2	*Compare r1 with r2 and jump over the next instruction if equal*
DEC r1	*Decrement register r1 by 1*
EOR r1,r2	*Exclusive bit wise Or register r1 with register r2 and store result in r1*
ICALL	*Call the subroutine at the address in register pair Z (ZH:ZL, R31:R30)*
IJMP IN r1,p1	*Jump to the address in register pair Z (ZH:ZL, R31:R30)*
INC r1	*Increment register r1 by 1*
LD r1,(rp,rp+,-rp)	*Load the register r1 with the content at the location that register pair rp (X, Y or Z) points to (rp+ increments the register pair after loading, -rp decrements the register pair prior to loading)*

LDD r1,ry+k63	Load the register r1 with the content at the location that register pair ry (Y or Z), displaced by the constant k63, points to
LDI rh,c255	Load the upper register rh with the constant c255
LDS r1,c65535	Load register r1 with the content at location c65535
LPM LPM r1 LPM r1,Z+ LPM r1,-Z	Load register R0 with the content of the flash memory at the location that register pair Z (ZH:ZL, R31:R30), divided by 2, points to, bit 0 in Z points to lower (0) or upper (1) byte in flash (Load register r1, Z+ increment Z after loading, -Z decrement Z prior to loading)
LSL r1	Logical shift left register r1
LSR r1	Logical shift right register r1
MOV r1,r2	Move register r2 to register r1
NEG r1	Subtract register r1 from Zero
NOP	No operation
OR r1,r2	Bit wise or register r1 with register r2 and store result in register r1
ORI rh,c255	Bit wise or the upper register r1 with the constant c255
OUT p1,r1	Copy register r1 to I/O port p1
POP r1	Increase the stack pointer and pop the last byte on stack to register r1
PUSH r1	Push register r1 to the stack and decrease the stack pointer
RCALL c4096	Push program counter on stack and add signed constant c4096 to the program counter (relative call)
RET	Pop program counter from stack (return to call address)
RETI	Enable interrupts and pop program counter from stack (return from interrupt)
R JMP c4096	Relative jump, add signed constant c4096 to program address
ROL r1	Rotate register r1 left, copy carry flag to bit 0
ROR r1	Rotate register r1 right, copy carry flag to bit 7
SBC r1,r2	Subtract r2 and the carry flag from register r1 and write result to r1
SBCI rh,c255	Subtract constant c255 and carry flag from the upper register rh and write result to rh
SBI pl,b7	Set bit b7 in the lower port pl
SBIC pl,b7	If bit b7 in the lower port pl is clear, jump over the next instruction
SBIS pl,b7	If bit b7 in the lower port pl is set, jump over the next instruction
SBIW rd,k63	Subtract the constant k63 from the register pair rd (rd+1:rd, rd = R24, R26, R28, R30)
SBR rh,c255	Set the bits in the upper register rh, that are one in constant c255
SBRC r1,b7	If bit b7 in register r1 is clear, jump over next instruction
SBRS r1,b7	If bit b7 in register r1 is set, jump over next instruction
SEC	Set carry flag in status register
SEH	Set half carry flag in status register
SEI	Set interrupt flag in status register, enable interrupt execution
SEN	Set negative flag in status register
SER rh	Set all bits in the upper register rh
SES	Set sign flag in status register
SET	Set transfer flag in status register
SEV	Set overflow flag in status register
SEZ	Set zero flag in status register
SLEEP	Put controller to the selected sleep mode
ST (rp/rp+/-rp),r1	Store content in register r1 to the memory location in register pair rp (rp = X, Y, Z; rp+: increment register pair after store; -rp: decrement register pair prior to store)
STD ry+k63,r1	Store the content of register r1 at the location that register pair ry (Y or Z), displaced by the constant k63, points to
STS c65535,r1	Store the content of register r1 at the location c65535
SUB r1,r2	Subtract register r2 from register r1 and write result to r1
SUBI rh,c255	Subtract the constant c255 from the upper register rh
SWAP r1	Exchange upper and lower nibble in register r1
TST r1	Compare register r1 with Zero
WDR	Watchdog reset

11.3 Port details

The table of the relevant ports in the ATMEL AVR types AT90S2313, 2323 and 8515. Byte wise accessible ports or register pairs are not displayed in detail. No warranty for correctness, see the original data sheets!

11.3.1 Status-Register, Accumulator flags

Port	*Function*	*Port-Address*	*RAM-Address*
SREG	Status Register Accumulator	0x3F	0x5F

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
I	T	H	S	V	N	Z	C

Bit	*Name*	*Meaning*	*Opportunities*	*Conmmand*
7	I	Global Interrupt Flag	0: Interrupts disabled	CLI
			1: Interrupts enabled	SEI
6	T	Bit storage	0: Stored bit is 0	CLT
			1: Stored bit is 1	SET
5	H	Halfcarry-Flag	0: No halfcarry occurred	CLH
			1: Halfcarry occurred	SEH
4	S	Sign-Flag	0: Sign positive	CLS
			1: Sign negative	SES
3	V	Two's complement-Flag	0: No carry occurred	CLV
			1: Carry occurred	SEV
2	N	Negative-Flag	0: Result was not negative/smaller	CLN
			1: Result was negative/smaller	SEN
1	Z	Zero-Flag	0: Result was not zero/unequal	CLZ
			1: Result was zero/equal	SEZ
0	C	Carry-Flag	0: No carry occurred	CLC
			1: Carry occurred	SEC

11.3.2 Stackpointer

Port	*Function*	*Port-Address*	*RAM-Address*
SPL/SPH	Stackpointer	003D/0x3E	0x5D/0x5E

Name	*Meaning*	*Availability*
SPL	Low-Byte of Stack pointer	From AT90S2313 upwards, not in 1200
SPH	High-Byte of Stack pointer	From AT90S8515 upwards, only in devices with >256 bytes internal SRAM

11.3.3 SRAM and External Interrupt control

Port	*Function*	*Port-Address*	*RAM-Address*
MCUCR	MCU General Control Register	0x35	0x55

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
SRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00

Bit	*Name*	*Meaning*	*Opportunities*
7	SRE	Ext. SRAM Enable	0=No external SRAM connected
			1=External SRAM connected
6	SRW	Ext. SRAM Wait States	0=No extra wait state on external SRAM
			1=Additional wait state on external SRAM
5	SE	Sleep Enable	0=Ignore SLEEP instructions
			1=SLEEP on instruction
4	SM	Sleep Mode	0=Idle Mode (Half sleep)
			1=Power Down Mode (Full sleep)

Bit	Name	Meaning	Opportunities
3	ISC11	Interrupt control Pin INT1 (connected to GIMSK)	00: Low-level initiates Interrupt
2	ISC10		01: Undefined
			10: Falling edge triggers interrupt
			11: Rising edge triggers interrupt
1	ISC01	Interrupt control Pin INT0 (connected to GIMSK)	00: Low-level initiates interrupt
0	ISC00		01: Undefined
			10: Falling edge triggers interrupt
			11: Rising edge triggers interrupt

11.3.4 External Interrupt Control

Port	Function	Port-Address	RAM-Address
GIMSK	General Interrupt Maskregister	0x3B	0x5B

7	6	5	4	3	2	1	0
INT1	INT0	-	-	-	-	-	-

Bit	Name	Meaning	Opportunities
7	INT1	Interrupt by external pin INT1 (connected to mode in MCUCR)	0: External INT1 disabled
			1: External INT1 enabled
6	INT0	Interrupt by external Pin INT0 (connected to mode in MCUCR)	0: External INT0 disabled
			1: External INT0 enabled
0...5	(Not used)		

Port	Function	Port-Address	RAM-Address
GIFR	General Interrupt Flag Register	0x3A	0x5A

7	6	5	4	3	2	1	0
INTF1	INTF0	-	-	-	-	-	-

Bit	Name	Meaning	Opportunities
7	INTF1	Interrupt by external pin INT1 occurred	Automatic clear by execution of the Int-Routine or Clear by instruction
6	INTF0	Interrupt by external pin INT0 occurred	
0...5	(Not used)		

11.3.5 Timer Interrupt Control

Port	Function	Port-Address	RAM-Address
TIMSK	Timer Interrupt Maskregister	0x39	0x59

7	6	5	4	3	2	1	0
TOIE1	OCIE1A	OCIE1B	-	TICIE1	-	TOIE0	-

Bit	Name	Meaning	Opportunities
7	TOIE1	Timer/Counter 1 Overflow-Interrupt	0: No Int at overflow 1: Int at overflow
6	OCIE1A	Timer/Counter 1 Compare A Interrupt	0: No Int at equal A 1: Int at equal A
5	OCIE1B	Timer/Counter 1 Compare B Interrupt	0: No Int at B 1: Int at equal B
4	(Not used)		
3	TICIE1	Timer/Counter 1 Capture Interrupt	0: No Int at Capture 1: Int at Capture
2	(Not used)		
1	TOIE0	Timer/Counter 0 Overflow-Interrupt	0: No Int at overflow 1: Int at overflow
0	(Not used)		

Port	Function	Port-Address	RAM-Address
TIFR	Timer Interrupt Flag Register	0x38	0x58

7	6	5	4	3	2	1	0
TOV1	OCF1A	OCF1B	-	ICF1	-	TOV0	-

Bit	Name	Meaning	Opportunities
7	TOV1	Timer/Counter 1 Overflow reached	Interrupt-Mode: Automatic Clear by execution of the Int-Routine OR Polling-Mode: Clear by instruction
6	OCF1A	Timer/Counter 1 Compare A reached	
5	OCF1B	Timer/Counter 1 Compare B reached	
4	(Not used)		
3	ICF1	Timer/Counter 1 Capture-Event occurred	
2	(not used)		
1	TOV0	Timer/Counter 0 Overflow occurred	
0	(not used)		

11.3.6 Timer/Counter 0

Port	Function	Port-Address	RAM-Address
TCCR0	Timer/Counter 0 Control Register	0x33	0x53

7	6	5	4	3	2	1	0
-	-	-	-	-	CS02	CS01	CS00

Bit	Name	Meaning	Opportunities
2..0	CS02..CS00	Timer Clock	000: Stop Timer 001: Clock = Chip clock 010: Clock = Chip clock / 8 011: Clock = Chip clock / 64 100: Clock = Chip clock / 256 101: Clock = Chip clock / 1024 110: Clock = falling edge of external Pin T0 111: Clock = rising edge of external Pin T0
3..7	(not used)		

Port	Function	Port-Address	RAM-Address
TCNT0	Timer/Counter 0 count register	0x32	0x52

11.3.7 Timer/Counter 1

Port	Function	Port-Address	RAM-Address
TCCR1A	Timer/Counter 1 Control Register A	0x2F	0x4F

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10

Bit	Name	Meaning	Opportunities
7	COM1A1	Compare Output A	00: OC1A/B not connected 01: OC1A/B changes polarity 10: OC1A/B to zero 11: OC1A/B to one
6	COM1A0		
5	COM1B1	Compare Output B	
4	COM1B0		
3	(not used)		
2			
1..0	PWM11 PWM10	Pulse width modulator	00: PWM off 01: 8-Bit PWM 10: 9-Bit PWM 11: 10-Bit PWM

Port	Function	Port-Address	RAM-Address
TCCR1B	Timer/Counter 1 Control Register B	0x2E	0x4E

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10

Bit	Name	Meaning	Opportunities
7	ICNC1	Noise Canceler on ICP-Pin	0: disabled, first edge starts sampling
			1: enabled, min four clock cycles
6	ICES1	Edge selection on Capture	0: falling edge triggers Capture
			1: rising edge triggers Capture
5..4	(not used)		
3	CTC1	Clear at Compare Match A	1: Counter set to zero if equal
2..0	CS12..CS10	Clock select	000: Counter stopped 001: Clock 010: Clock / 8 011: Clock / 64 100: Clock / 256 101: Clock / 1024 110: falling edge external Pin T1 111: rising edge external Pin T1

Port	Function	Port-Address	RAM-Address
TCNT1L/H	Timer/Counter 1 count register	0x2C/0x2D	0x4C/0x4D

Port	Function	Port-Address	RAM-Address
OCR1AL/H	Timer/Counter 1 Output Compare register A	0x2A/0x2B	0x4A/0x4B hex

Port	Function	Port-Address	RAM-Address
OCR1BL/H	Timer/Counter 1 Output Compare register B	0x28/0x29	0x48/0x49

Port	Function	Port-Address	RAM-Address
ICR1L/H	Timer/Counter 1 Input Capture Register	0x24/0x25	0x44/0x45

11.3.8 Watchdog-Timer

Port	Function	Port-Address	RAM-Address
WDTCR	Watchdog Timer Control Register	0x21	0x41

7	6	5	4	3	2	1	0
-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0

Bit	Name	Meaning	WDT-cycle at 5.0 Volt
7..5	(not used)		
4	WDTOE	Watchdog Turnoff Enable	Previous set to disabling of WDE required
3	WDE	Watchdog Enable	1: Watchdog active
2..0	WDP2..WDP0	Watchdog Timer Prescaler	000: 15 ms 001: 30 ms 010: 60 ms 011: 120 ms 100: 240 ms 101: 490 ms 110: 970 ms 111: 1,9 s

11.3.9 EEPROM

Port	Function	Port-Address	RAM-Address
EEARL/H	EEPROM Address Register	0x1E/0x1F	0x3E/0x3F

EEARH only in types with more than 256 Bytes EEPROM (from AT90S8515 upwards)

Port	Function	Port-Address	RAM-Address
EEDR	EEPROM Data Register	0x1D	0x3D

Port	Function	Port-Address	RAM-Address
EECR	EEPROM Control Register	0x1C	0x3C

7	6	5	4	3	2	1	0
-	-	-	-	-	EEMWE	EEWE	EERE

Bit	Name	Meaning	Function
7..3	(not used)		
2	EEMWE	EEPROM Master Write Enable	Previous set enables write cycle
1	EEWE	EEPROM Write Enable	Set to initiate write
0	EERE	EEPROM Read Enable	Set initiates read

11.3.10 Serial Peripheral Interface SPI

Port	Function	Port-Address	RAM-Address
SPCR	SPI Control Register	0x0D	0x2D

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Bit	Name	Meaning	Function
7	SPIE	SPI Interrupt Enable	0: Interrupts disabled
			1: Interrupts enabled
6	SPE	SPI Enable	0: SPI disabled
			1: SPI enabled
5	DORD	Data Order	0: MSB first
			1: LSB first

4	MSTR	Master/Slave Select	0: Slave
			1: Master
3	CPOL	Clock Polarity	0: Positive Clock Phase
			1: Negative Clock Phase
2	CPHA	Clock Phase	0: Sampling at beginning of Clock Phase
			1: Sampling at end of Clock Phase
1	SPR1	SCK clock frequency	00: Clock / 4
0	SPR0		01: Clock / 16
			10: Clock / 64
			11: Clock / 128

Port	*Function*	*Port-Address*	*RAM-Address*
SPSR	SPI Status Register	0x0E	0x2E

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
SPIF	WCOL	-	-	-	-	-	-

Bit	*Name*	*Meaning*	*Function*
7	SPIF	SPI Interrupt Flag	Interrupt request
6	WCOL	Write Collision Flag	Write collission occurred
5..0	(not used)		

Port	*Function*	*Port-Address*	*RAM-Address*
SPDR	SPI Data Register	0x0F	0x2F

11.3.11 UART

Port	*Function*	*Port-Address*	*RAM-Address*
UDR	UART I/O Data Register	0x0C	0x2C

Port	*Function*	*Port-Address*	*RAM-Address*
USR	UART Status Register	0x0B	0x2B

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
RXC	TXC	UDRE	FE	OR	-	-	-

Bit	*Name*	*Meaning*	*Function*
7	RXC	UART Receive Complete	1: Char received
6	TXC	UART Transmit Complete	1: Shift register empty
5	UDRE	UART Data Register Empty	1: Transmit register available
4	FE	Framing Error	1: Illegal Stop-Bit
3	OR	Overrun	1: Lost char
2..0	(not used)		

Port	*Function*	*Port-Address*	*RAM-Address*
UCR	UART Control Register	0x0A	0x2A

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

Bit	*Name*	*Meaning*	*Function*
7	RXCIE	RX Complete Interrupt Enable	1: Interrupt on received char
6	TXCIE	TX Complete Interrupt Enable	1: Interrupt at transmit complete
5	UDRIE	Data Register Empty Interrupt Enable	1: Interrupt on transmit buffer empty

4	RXEN	Receiver Enable	1: Receiver enabled
3	TXEN	Transmitter Enable	1: Transmitter enabled
2	CHR9	9-bit Characters	1: Char length 9 Bit
1	RXB8	Receive Data Bit 8	(holds 9th data bit on receive)
0	TXB8	Transmit Data Bit 8	(write 9th data bit for transmit here)

Port	*Function*	*Port-Address*	*RAM-Address*
UBRR	UART Baud Rate Register	0x09	0x29

11.3.12 Analog Comparator

Port	*Function*	*Port-Address*	*RAM-Address*
ACSR	Analog Comparator Control and Status Register	0x08	0x28

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Bit	*Name*	*Meaning*	*Function*
7	ACD	Disable	Disable Comparators
6	(not used)		
5	ACO	Comparator Output	Read: Output of the Comparators
4	ACI	Interrupt Flag	1: Interrupt request
3	ACIE	Interrupt Enable	1: Interrupts enabled
2	ACIC	Input Capture Enable	1: Connect to Timer 1 Capture
1	ACIS1	Input Capture Enable	00: Interrupt on edge change
			01: (not used)
0	ACIS0		10: Interrupt on falling edge
			11: Interrupt on rising edge

11.3.13 I/O Ports

Port	*Register*	*Function*	*Port-Address*	*RAM-Address*
A	PORTA	Data Register	0x1B	0x3B
	DDRA	Data Direction Register	0x1A	0x3A
	PINA	Input Pins Address	0x19	0x39
B	PORTB	Data Register	0x18	0x38
	DDRB	Data Direction Register	0x17	0x37
	PINB	Input Pins Address	0x16	0x36
C	PORTC	Data Register	0x15	0x35
	DDRC	Data Direction Register	0x14	0x34
	PINC	Input Pins Address	0x13	0x33
D	PORTD	Data Register	0x12	0x32
	DDRD	Data Direction Register	0x11	0x31
	PIND	Input Pins Address	0x10	0x30

11.4 Ports, alphabetic order

ACSR, Analog Comparator Control and Status Register
DDRx, Port x Data Direction Register
EEAR, EEPROM address Register
EECR, EEPROM Control Register
EEDR, EEPROM Data Register
GIFR, General Interrupt Flag Register
GIMSK, General Interrupt Mask Register
ICR1L/H, Input Capture Register 1
MCUCR, MCU General Control Register
OCR1A, Output Compare Register 1 A

OCR1B, Output Compare Register 1 B
PINx, Port Input Access
PORTx, Port x Output Register
SPL/SPH, Stackpointer
SPCR, Serial Peripheral Control Register
SPDR, Serial Peripheral Data Register
SPSR, Serial Peripheral Status Register
SREG, Status Register
TCCR0, Timer/Counter Control Register, Timer 0
TCCR1A, Timer/Counter Control Register 1 A
TCCR1B, Timer/Counter Control Register 1 B
TCNT0, Timer/Counter Register, Counter 0
TCNT1, Timer/Counter Register, Counter 1
TIFR, Timer Interrupt Flag Register
TIMSK, Timer Interrupt Mask Register
UBRR, UART Baud Rate Register
UCR, UART Control Register
UDR, UART Data Register
WDTCR, Watchdog Timer Control Register

11.5 List of abbreviations

The abbreviations used are chosen to include the value range. Register pairs are named by the lower of the two registers. Constants in jump instructions are automatically calculated from the respective labels during assembly.

Category	*Abbrev.*	*Means ...*	*Value range*
Register	r1	Ordinary Source and Target register	R0..R31
	r2	Ordinary Source register	
	rh	Upper page register	R16..R31
	rd	Twin register	R24(R25), R26(R27), R28(R29), R30(R31)
	rp	Pointer register	X=R26(R27), Y=R28(R29), Z=R30(R31)
	ry	Pointer register with displacement	Y=R28(R29), Z=R30(R31)
Constant	k63	Pointer-constant	0..63
	c127	Conditioned jump distance	-64..+63
	c255	8-Bit-Constant	0..255
	c4096	Relative jump distance	-2048..+2047
	c65535	16-Bit-Address	0..65535
Bit	b7	Bit position	0..7
Port	p1	Ordinary Port	0..63
	pl	Lower page port	0..31